ORSAY
no d'ordre: **3502**

UNIVERSITÉ DE PARIS–SUD

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

LE TITRE DE DOCTEUR 3ème CYCLE

SPECIALITÉ: INFORMATIQUE

par

Valérie MIGOT

SUJET:

UN PASCAL MODULAIRE SOUS MENTOR

Soutenue le 22 septembre 1983 devant la Commission d'examen composée de:

MM.	G.GUIHO	Président
	F.BANCILHON	Examinateurs
	M.C.GAUDEL	
	G.KAHN	
	F.PRUSKER	

UNIV.
LEIDEN
BIBL.

ORSAY
no d'ordre: **3502**

UNIVERSITÉ DE PARIS–SUD

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

LE TITRE DE DOCTEUR 3ème CYCLE

SPECIALITÉ: INFORMATIQUE

par

Valérie MIGOT

SUJET:

UN PASCAL MODULAIRE SOUS MENTOR

Soutenue le 22 septembre 1983 devant la Commission d'examen composée de:

MM.	G.GUIHO	Président
	F.BANCILHON	Examinateurs
	M.C.GAUDEL	
	G.KAHN	
	F.PRUSKER	

Je remercie Monsieur G.GUIHO d'avoir accepté de présider ce jury, Madame M.C.GAUDEL et Messieurs F.BANCILHON, G.KAHN et F.PRUSKER d'avoir bien voulu participer à ce jury.

Je remercie tout particulièrement A.M.VERCOUSTRE qui m'a introduite dans le projet LEGOS à l'INRIA, J.L.BOUCHENEZ, F.PRUSKER qui m'ont suggéré ce travail et M.LOYER qui m'a apporté continuellement aide et conseils.

Merci enfin à G.KAHN, B.LANG et B.MELESE qui m'ont fait découvrir MENTOR, sans lequel cette thèse n'aurait pas pu voir le jour.

PAPIER RÉCUPÉRÉ ET RECYCL

Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41547306_0040

ABSTRACT

This thesis presents some extensions to the Pascal Language which give facilities for separate compilation and modularity. These extensions have been realized with the Mentor Pascal Program Manipulation System. A pre-processor transforms programs written in extended Pascal into programs written in standard Pascal. The pre-processor allows to define interfaces of a programs (i.e coherent set of exported objects). These interfaces are usable by other programs of the application. A catalogue manages the different components of the application (programs and interfaces) and detects the consequences of a modification of a component for the rest of the application.

Table des matières

Introduction

1 Présentation des extensions au Pascal standard
- 1.1 Définitions
- 1.2 Les ajouts au Pascal standard
  - 1.2.1 Les ajouts
  - 1.2.2 Les avantages
  - 1.2.3 Les répercussions
- 1.3 Un mécanisme complémentaire: les interfaces réduites
  - 1.3.1 Interfaces réduites
    - 1.3.1.1 Interface réduite sur un composant
    - 1.3.1.2 Interface sur un ensemble de composants
  - 1.3.2 Le constructeur d'interfaces

2 Contexte de réalisation du pré-processeur
- 2.1 Le compilateur Pascal-Multics
- 2.2 Présentation rapide de Mentor
- 2.3 Adaptation des mécanismes
  - 2.3.1 Définitions
  - 2.3.2 Adaptation
  - 2.3.3 La fusion structurelle

3 Utilisation du pré-processeur sous Mentor
- 3.1 Le catalogue
- 3.2 Définition du contexte de compilation
- 3.3 Compilation d'un programme modulaire
- 3.4 Opérations sur les interfaces: réduction et fusion
  - 3.4.1 Réduction d'une interface
  - 3.4.2 Fusion de plusieurs interfaces
- 3.5 Interface virtuelle

4 Description technique du pré-processeur
- 4.1 Le catalogue

4.1.1 La liste USE
4.1.2 La structure du catalogue
4.2 Description détaillée des commandes du pré-processeur
4.2.1 La commande use
4.2.2 La commande mcompile
4.2.3 La commande crintf
4.2.4 La commande iface
4.2.5 La commande reduc
4.2.6 La commande metdef
4.2.7 La commande merge
4.2.8 La commande saveintf
4.2.9 La commande ecrit
4.2.10 La commande fusionintf
4.2.11 La commande realize
4.2.12 La commande checkintf
4.2.13 La commande checkuseintf
4.2.14 La commande querecomp
4.3 Remarques diverses
4.3.1 Sauvegarde des programmes et des interfaces
4.3.2 Choix des noms des interfaces et des programmes compilables
4.3.3 Le traitement des messages
4.3.4 La documentation du pré-processeur
4.3.5 Problème lié au compilateur Pascal-Multics
4.3.6 Problèmes posés par les fichiers
4.3.7 Un Pascal Modulaire avec Métal

Conclusion

Annexes
1 Session dans l'environnement Pascal Modulaire sous Mentor
2 Les programmes Mentol
3 Les programmes Pascal

Bibliographie

Introduction

Cette thèse présente les extensions que nous avons faites au langage Pascal pour introduire la compilation séparée et la modularité. Ce travail a été réalisé dans le cadre du projet LEGOS à l'INRIA. Les extensions ont été faites à l'aide de l'environnement de programmation Mentor-Pascal.

Le choix de ces extensions a été guidé par les objectifs suivants:

- permettre la réalisation d'une application Pascal au moyen de plusieurs unités de compilation en assurant la cohérence globale, c'est-à-dire en garantissant que les objets définis dans une unité seront manipulés de façon correcte dans les autres unités de compilation de l'application,

- offrir des mécanismes permettant d'expliciter les liens entre les différentes unités de compilation,

- limiter au minimum les ajouts au Pascal standard ([ISO 82] [AFN 82]) et ne pas compliquer par ces ajouts les mécanismes nécessaires à la compilation d'un programme Pascal.

Ces extensions sont dérivées des mécanismes définis pour le langage Legos ([BOU 81] [LOY 81]).

Elles sont traitées sous Mentor par un pré-processeur qui transforme un programme écrit en Pascal modulaire en un programme acceptable par un compilateur standard.

En respectant ces objectifs, nous ne prétendons ni définir de nouveaux mécanismes de modularité ni, a fortiori, un nouveau langage, mais nous nous situons plutôt dans le cadre des propositions d'extension de Pascal demandées par des organismes comme l'ISO, l'AFNOR ou l'ANSI.

La première partie de cette thèse présente les extensions faites au Pascal standard ISO pour introduire la compilation séparée et des mécanismes de modularité. Nous décrivons ensuite le contexte de réalisation du pré-processeur, puis son utilisation sous Mentor. La dernière partie est une description technique de son implémentation en Mentol.

1– Présentation des extensions au Pascal standard

1.1- Définitions

L' *unité de compilation* est le programme Pascal tel qu'il est défini dans la norme ISO.

Une *application* est constituée d'une ou plusieurs unités de compilation.

Nous appelons *interface* d'un programme l'ensemble des objets déclarés au plus haut niveau dans un programme.

Enfin le terme *composant* nous sert à désigner interface ou programme lorsqu'il n'y a pas lieu de distinguer ces deux catégories d'objets.

Dans un programme P, il est possible de référencer des objets déclarés dans d'autres programmes. Ces programmes -ou plus exactement les interfaces de ces programmes- constituent le *contexte* du programme P.
Ces programmes sont indiqués dans une liste *USE* placée en tête du programme P.

On a alors la règle suivante:

> Tout objet référencé dans un programme P doit avoir été déclaré dans le programme P ou dans une des interfaces des programmes qui appartiennent à son contexte.

Exemple de programmes écrits en Pascal modulaire:

```
program DEFINITION;
     const
          MAX     =20;
     type PILE    =
               record
                    P:array[1..MAX]of INTEGER;
                    TOP:INTEGER
               end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function SOMMET(var PIL:PILE):INTEGER;external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;
```

```
procedure INIT(var PIL:PILE);external;
```

Le corps de chaque sous-programme, dont l'en-tête est déclaré dans le programme DEFINITION, est défini dans le programme suivant:

```
use DEFINITION;
program REALISATION;

     procedure EMPILER;
          begin
          if PIL.TOP<MAX then
               begin
               PIL.TOP:=PIL.TOP+1;
               PIL.P[PIL.TOP]:=ITEM
               end
          end;

     procedure DEPILER;
          begin
          if PIL.TOP#0 then PIL.TOP:=PIL.TOP-1
          end;

     function SOMMET;
          begin
          if PIL.TOP#0 then SOMMET:=PIL.P[PIL.TOP]
          else SOMMET:=-1234567890
          end;

     function PILENONVIDE;
          begin
          if PIL.TOP=0 then PILENONVIDE:=FALSE
          else PILENONVIDE:=TRUE
          end;

     procedure INIT;
          begin
          PIL.TOP:=0
          end;
```

Ce troisième programme référence le programme DEFINITION:

```
use DEFINITION;
program UTILISATION(OUTPUT);
     var  PILE1,PILE2:PILE;
          I:INTEGER;
     begin
     INIT(PILE1);
     INIT(PILE2);
     for I:=1 to 5 do EMPILER(PILE1,I);
     while PILENONVIDE(PILE1)do
          begin
          EMPILER(PILE2,-SOMMET(PILE1));
          DEPILER(PILE1)
          end;
     while PILENONVIDE(PILE2)do
          begin
```

```
        WRITELN(SOMMET(PILE2));
        DEPILER(PILE2)
        end
end.
```

Ces extensions se rapprochent de l'utilisation des modules en Mesa [MIT 79]. En effet, Mesa distingue deux sortes de modules: les modules de définition et les modules de réalisation. Les modules de définition ou interfaces contiennent des spécifications d'objets et servent pour la compilation d'autres modules (modules de réalisation ou modules de définition plus élaborés). Ils contiennent des déclarations virtuelles d'objets qui doivent être exportés par un module de réalisation. Dans les modules de réalisation se trouvent les déclarations effectives des objets décrits dans un module de définition. Seuls les modules de réalisation existent à l'exécution. Les modules de définition ne servent qu'à décrire les liens entre modules de réalisation et, pour une application, ils "disparaissent" après la phase d'édition de liens.

Il n'y a pas - au niveau langage - un lien unique entre module de définition et module de réalisation. Ainsi, à un module de définition peuvent correspondre plusieurs modules de réalisation. Inversement, plusieurs modules de définition peuvent être associés à un même module de réalisation. Les choix entre les différents liens se font à l'aide d'un mécanisme complémentaire (appelé langage de connexion) au moment de l'édition de liens.

Cette approche est différente de celle d'Ada [ADA 80] ou - au niveau langage - il y a un lien unique entre spécification et réalisation. A la spécification d'un package A correspond la réalisation d'un package body de nom A.

La solution que nous avons adoptée est semblable à celle de Mesa et suppose un mécanisme complémentaire, comme en Mesa, afin d'associer réalisation et spécification et d'effectuer, à l'édition de liens, les connexions entre les composants. Ce mécanisme n'est pas décrit ici; on trouvera un exemple de ce qu'il peut être en Legos [BOU 81] [LOY 81].

1.2- Les ajouts au Pascal standard

1.2.1- Les ajouts

Les ajouts proposés sont minimes. Ils consistent en deux points:

1- Ajouter en tête d'un programme une liste *USE* énumérant les autres unités de compilation dont des objets sont référencés dans le programme P.

2- Autoriser que l'en-tête et le corps d'un sous-programme de plus haut niveau soient dans deux unités de compilation distinctes.

Dans l'unité de compilation contenant la définition de l'en-tête, le corps est alors remplacé par la directive EXTERNAL. Cette directive EXTERNAL est semblable à la directive FORWARD, la seule différence étant que le corps associé n'est pas défini ultérieurement dans la même

unité de compilation mais dans une autre unité.

Quant à l'unité contenant la définition du corps associé, elle doit contenir dans sa liste *USE* le nom de l'unité où a été défini l'en-tête.

Nous avons refusé de faire d'autres ajouts comme d'introduire un mécanisme pour cacher la structure d'un type à l'extérieur de l'unité de compilation où il a été défini ou comme d'étendre la notation pointée sous la forme: nom_de_programme.nom_d'objet .

Ces ajouts supplémentaires, bien que fort utiles, nous auraient trop éloignés du Pascal standard et auraient rendu plus compliquée l'adaptation de compilateurs existants.

C'est ce même souci de simplicité qui nous a conduit à ne pas autoriser la compilation séparée des corps des sous-programmes imbriqués comme c'est le cas en Ada [ADA 80] et dans la proposition d'extension à Pascal, de D.C.Robbins et L.R.Carter [CAR 82].

1.2.2- Les avantages

Ces ajouts offrent les avantages suivants:

- permettre la vérification de cohérence (en particulier la cohérence des types) entre les différents composants d'une application,

- éviter la redescription des objets externes dans les composants qui les importent, ce qui est en général fastidieux et source d'erreurs,

- connaître facilement, grâce aux listes *USE,* l'ensemble des dépendances entre les composants d'une application et ainsi mesurer les conséquences entraînées par la modification d'un composant sur le reste de l'application (en particulier, déduire les recompilations).

1.2.3- Les répercussions

Un programme ne peut être compilé que si l'ensemble des programmes constituant son contexte a été compilé au préalable. Cela définit donc un ordre partiel de compilation entre les unités d'une application. De même la modification d'une unité de compilation P implique normalement la recompilation des unités référençant P.

La gestion des divers composants d'une application et le maintien de la cohérence de l'ensemble sont faits à l'aide d'un catalogue réunissant les informations contenues dans les listes *USE.* Parmi d'autres fonctionnalités, le catalogue doit pouvoir donner à l'utilisateur une liste ordonnée des différents composants qu'il faut recompiler pour mettre l'application dans un état cohérent. Une solution plus élaborée, s'il existe un outil reconfigurateur comme Make [FEL 77], consiste, pour le catalogue, à pouvoir produire un fichier de commandes pour cet outil.

Pour avoir un compilateur traitant les mécanismes de compilation séparée proposés, les ajouts à faire consistent:

- en fin de compilation, à conserver une table de description des objets déclarés au plus haut niveau dans le programme P, c'est-à-dire à conserver sous forme interne l'interface de P;

- en début de compilation, à fusionner les tables de description des programmes figurant dans la liste *USE* de P, c'est-à-dire à bâtir le contexte de P.

Ces deux opérations se situant en amont et en aval de la compilation proprement dite peuvent être assez facilement ajoutées à un compilateur Pascal standard.

Si l'on dispose d'un compilateur Pascal standard comprenant des mécanismes de compilation séparée élémentaires (def, ref), une autre solution consiste à transformer, à l'aide d'un pré-processeur, les textes des programmes écrits en Pascal modulaire en programmes Pascal acceptés par ce compilateur. C'est la solution que nous avons adoptée dans notre réalisation sous Mentor.

Remarque:

Pour des raisons d'efficacité, il est important que le contexte de compilation d'un programme soit réduit au minimum. Pour cela nous n'avons pas autorisé l'héritage de listes *USE* entre unités de compilation. Ainsi, si un programme Q est mentionné dans la liste *USE* d'un programme R et si un programme P est lui-même mentionné dans celle de Q, alors P ne fait pas implicitement partie du contexte de R. Et P n'aura à être mis dans la liste *USE* de R que si R référence effectivement des objets de P.

Ceci permet d'une part de ne garder trace, au moyen des listes *USE,* que des liaisons effectives entre programmes et d'autre part de minimiser la taille de la table de contexte résultant de la fusion.

1.3- Un mécanisme complémentaire: les interfaces réduites

1.3.1- Interfaces réduites

Une autre opération permettant de limiter les contextes de compilation consiste à pouvoir définir une ou plusieurs *interfaces réduites* à partir de ce que nous avons appelé jusqu'ici interface d'un programme et que nous appellerons désormais *interface globale* d'un programme (i.e. l'ensemble des objets déclarés au plus haut niveau).

1.3.1.1- Interface réduite sur un composant

Une interface réduite est obtenue en ôtant certaines déclarations de l'interface globale d'un programme (ou d'une interface déjà réduite). Cette réduction doit se faire en respectant la règle suivante:

Si, lors de la création d'une interface réduite I sur un composant C, une déclaration D est conservée, alors tout objet utilisé pour la définition de D et déclaré dans C doit lui aussi être conservé dans I.

Cette règle garantit que les objets conservés dans l'interface forment un ensemble cohérent. Ainsi, si une déclaration de sous-programme est conservée, on devra conserver également les déclarations des types des paramètres formels.

A chaque interface réduite est associé un nom qui peut être mentionné dans la liste *USE* d'un autre programme.

Pour un même programme, comme dans Protel [CAS 81], on peut définir plusieurs interfaces réduites: chacune est alors un "cache" particulier donnant un certain "angle de vue" sur le programme.

1.3.1.2- Interface sur un ensemble de composants

Une interface réduite peut être construite, de la même façon, non plus sur un composant, mais sur un ensemble de composants. Ce mécanisme permet de définir une interface plus synthétique pour un ensemble de composants constituant une sous-partie d'une application.

Pour l'utilisateur de cette interface, la façon dont sont agencés les composants constituant cette sous-partie est cachée. Pour lui, tout se passe comme si cette interface ne portait que sur un seul composant.

Ceci permet une description hiérarchique d'une application. L'application ou système se décompose en plusieurs sous-systèmes, caractérisés par leurs interfaces. Chaque sous-système se décompose à son tour en composants plus élémentaires.

Ce mécanisme s'inspire des configurations définies dans le langage Mesa [MIT 79].

1.3.2- Le constructeur d'interfaces

D'un point de vue pratique, la réduction d'une interface globale consiste à produire une table réduite à partir des tables de descriptions des objets déclarés dans des programmes déjà compilés.

Cette réduction est réalisée par un processeur distinct du compilateur, le constructeur d'interfaces.

Ce constructeur peut être conçu pour travailler à partir d'une description de l'interface réduite fournie par l'utilisateur sous une forme syntaxique proche de Pascal. Le constructeur, tout en produisant les tables réduites, vérifie la cohérence de cette description. C'est l'approche choisie pour le langage Legos [BOU 81].

Une autre solution consiste à définir un constructeur d'interfaces interactif. Le programmeur se contente d'indiquer les noms des objets qu'il souhaite conserver dans l'interface réduite et est guidé dans ces choix par le constructeur. Celui-ci contrôle à chaque instant la correction de l'interface en construction et interdit les opérations illicites. Outre les tables réduites, le constucteur produit le texte de l'interface construite sous forme Pascal.

Cette solution a été retenue pour la réalisation du pré-processeur que nous allons maintenant présenter.

2– Contexte de réalisation du pré–processeur

Pour implémenter, sous le système Multics de l'INRIA, les mécanismes présentés dans la première partie, il était possible d'écrire de façon classique un pré-processeur au compilateur. Nous avons préféré utiliser Mentor pour programmer ce pré-processeur et les commandes de manipulation des programmes écrits dans notre Pascal-modulaire.

2.1- Le compilateur Pascal-Multics

Le compilateur Pascal disponible sur Multics, dans la version 6.02, est muni de mécanismes de compilation séparée élémentaires.

Il permet de partager des variables et des sous-programmes entre des programmes compilés séparément.

Ces mécanismes se réduisent à l'importation et à l'exportation de variables et de sous-programmes en respectant les règles suivantes [PAS 81]:

- Dans le programme qui exporte,

les variables et les sous-programmes exportés doivent être déclarés au plus haut niveau avec l'attribut 'def'.

D'autre part, si le corps du programme est vide, dans l'en-tête, le nom du programme doit être précédé de 'def'.

- Dans le programme qui importe,

les noms des variables et sous-programmes importés doivent être cités dans une liste, appelée liste d'externes, située dans l'en-tête du programme.

De plus, les objets importés doivent être redéclarés dans le programme qui les importe (pour les sous-programmes, le corps est remplacé par la directive 'external').

2.2- Présentation rapide de Mentor

Mentor est un système de manipulation de programmes représentés par des arbres de syntaxe abstraite ([DON 75] [DON 79] [DON 80] [DON 81] [MEL 80] [MEL 81]). Les données manipulées sont des fragments de programmes et des informations associées à ces programmes, comme par exemple les commentaires. Ces informations portent le nom d'annotations structurées. Les données ont une représentation abstraite unique sous forme d'arbres étiquetés.

Le langage de commande du système est un langage interactif appelé Mentol. Il permet de mettre en oeuvre divers outils de manipulation des représentations abstraites et de développer ainsi un environnement de programmation.

Les outils de base incluent les primitives de repérage et de manipulation (destruction ou insertion de sous-arbres par exemple) et des processeurs spécialisés (en particulier analyseurs et décompilateurs permettant la traduction entre les arbres de la représentation abstraite et le texte source). Les variables de Mentor sont des repères dans les arborescences de programme. Leur nom est de la forme: @<identificateur>. Un repère standard, dit repère courant, est utilisé implicitement dans la plupart des manipulations. Le nom explicite de ce repère est @k. Il est également appelé programme courant ou variable courante.

L'impression d'un sous-arbre est obtenue par la commande P. @T P N imprime le sous-arbre désigné par la variable @T avec un niveau de détail indiqué par l'entier N. Le texte du programme est éventuellement abrégé à l'aide des symboles '#' pour les sous-arbres dont l'opérateur de tête est d'arité fixe et '...' pour les listes.

Le langage Mentol permet d'écrire des procédures de manipulation et de transformation d'arbres. L'appel d'une procédure se fait en indiquant son nom précédé d'un point et suivi de ses arguments éventuels entre les symboles '<' et '>'.

Sous Mentor, un programme peut être sauvegardé sous deux formes. Le fichier dont le nom est suffixé par *polish* contient la forme arborescente du programme. Celui dont le nom est suffixé par *pascal* est un fichier texte traditionnel.

Nous avons utilisé Mentor dans la version 4.01 appliquée au langage Pascal. Pour utiliser le pré-processeur, il faut se placer au début de la session dans l'environnement pascal-modulaire; celui-ci se compose d'un ensemble de procédures Mentol permettant les transformations de programmes et la gestion d'un catalogue.

Toute la session doit se dérouler dans le même directory. Le pré-processeur est ainsi indépendant du système Multics et facilement portable.

2.3- Adaptation des mécanismes

Avant de présenter le pré-processeur, nous allons donner quelques définitions, justifier les adaptations que nous avons dû faire et expliquer le mécanisme de base utilisé dans la plupart des manipulations effectuées sur les programmes.

2.3.1- Définitions

Nous appelons *programme Pascal–modulaire* ou *programme modulaire* un programme écrit dans le langage Pascal complété par les mécanismes de modularité que nous avons précédemment décrits.

Un *programme Pascal–compilable* ou *programme compilable* est un programme acceptable par le compilateur Pascal disponible sur Multics.

Une *interface cohérente* est une interface dont tous les objets utilisés dans ses déclarations sont décrits dans l'interface ou appartiennent à son contexte.

2.3.2- Adaptation

Les objets manipulés sous Mentor, dans la version utilisée, devant être des "morceaux" de programme Pascal, nous avons dû adapter les mécanismes présentés dans la première partie de la façon suivante:

La liste *USE* est introduite à l'aide d'une commande, sous forme d'une annotation structurée, située en tête du programme. Cette annotation apparaît comme un commentaire dans le texte du programme décompilé:

```
 (*USE(UTIL,LIST)*)
program START;
     ...
     begin
     .
     .
     end.
```

Les interfaces devant avoir la même structure syntaxique qu'un programme pour être acceptables par Mentor, les en-têtes de sous-programmes apparaissant dans les interfaces sont suivis de la directive 'external'.

D'autre part, le compilateur interdit la séparation dans deux programmes distincts de l'en-tête et du corps d'un sous-programme. Il ne permet donc pas la décomposition d'une unité fonctionnelle en deux unités de compilation correspondant à la spécification et la réalisation.

Pour pallier cette difficulté, nous avons défini la notion d'interface virtuelle: une interface virtuelle contient des descriptions d'objets (constantes, types, variables) et des en-têtes de sous-programmes dont les corps sont définis ultérieurement dans un autre programme. Une interface virtuelle se présente comme une interface classique.

```
(*USE*)
program LISTDEF;
     type LISTP   =@LISTREC;
          LISTREC =
               record
                    VALU:INTEGER;
                    NEXT:LISTP
               end;

     procedure INSERTION(var HEAD:LISTP;VALEUR:INTEGER);external;
```

Elle permet de définir une spécification pour une partie d'une application non encore écrite; elle sert ainsi de cahier des charges pour ceux qui doivent réaliser cette spécification et peut également être utilisée pour la compilation d'autres programmes, sans nécessiter que la réalisation soit déjà écrite. L'interface virtuelle et le programme la réalisant sont transformés par le pré-processeur en un seul programme compilable.

Voici un programme réalisant l'interface précédente:

```
(*USE(LISTDEF)*)
program LISTREALIS;

     procedure INSERTION;
          var  PT:LISTP;
          begin
          PT:=HEAD;
          NEW(HEAD);
          with HEAD@ do
               begin
               VALU:=INTEGER;
               NEXT:=PT
               end
          end;
```

Remarquons qu'à la différence de Mesa, nous n'obligeons pas à redéclarer dans le programme les objets de l'interface virtuelle.

2.3.3- La fusion structurelle

L'opération de base du pré-processeur consiste à fusionner deux programmes Pascal. Cette opération est utilisée lors de la création du programme compilable (il faut fusionner les interfaces utilisées dans le programme modulaire). Lorsque l'on veut obtenir une interface sur un ensemble d'interfaces, il faut également fusionner les interfaces considérées. Dans les deux cas, le mécanisme élémentaire de fusion est itéré pour chaque interface.

Pour obtenir un programme syntaxiquement correct par fusion de deux programmes, il faut faire une inclusion structurée de texte. Sous Mentor, cela revient à insérer des sous-arbres. Pour réaliser cette opération, on suppose que l'un des deux programmes a un corps vide, ce qui est légitime car l'un des deux programmes est toujours une interface pour ce qui nous concerne.

Soient P1 et P2 les deux programmes que l'on veut fusionner. On suppose que P1 a un corps vide. Le programme P3 résultant de la fusion de P1 avec P2 est obtenu en dupliquant P2 dans P3, puis en insérant les déclarations de constantes de P1, s'il y en a, au début de la zone de constantes de P3 si elle existe sinon en créant une zone de constantes dans P3 identique à celle de P1. On opère de même avec les déclarations de types, puis celles de variables et finalement avec les sous-programmes. Cette insertion se fait sans modifier l'ordre dans lequel sont déclarés les objets dans chacun des programmes.

Dans le programme P3, les déclarations contenues dans P1 apparaissent toujours avant celles contenues dans P2 dans chacune des quatre zones de constantes, types, variables et sous-programmes.

3– Utilisation du pré–processeur sous Mentor

Pour utiliser le pré-processeur, il faut charger l'environnement Pascal-modulaire au début de la session Mentor. On dispose alors des commandes suivantes:

pcat
: imprime le catalogue courant

chcat
: change le catalogue courant

use
: positionne la liste *USE* du programme courant

mcompile
: compile un programme modulaire

crintf
: crée une interface à partir d'un programme ou d'une interface

fusionintf
: crée une interface à partir de plusieurs interfaces

saveintf
: sauvegarde une interface et range son nom dans le catalogue

realize
: permet d'indiquer qu'un programme réalise une interface virtuelle et crée le programme modulaire correspondant

D'autres commandes sont disponibles; elles sont décrites dans la section 4.

3.1- Le catalogue

A chaque application correspond un catalogue qui contient tous les renseignements sur sa structure, c'est-à-dire les noms des programmes modulaires et des interfaces constituant l'application.

Ces éléments sont rangés automatiquement dans le catalogue lors de l'exécution des commandes de compilation et de création d'interfaces, chaque fois qu'un composant a été créé ou modifié. Les renseignements contenus dans le catalogue sont utilisés par les commandes constituant le pré-processeur.

A chaque nom d'interface sont associés:
- la liste *USE* de l'interface
- le nom du programme dont elle est l'interface, s'il existe
- éventuellement, le nom des interfaces qu'elle englobe
- la date de la dernière modification.

A chaque nom de programme sont associées:
- la liste *USE* du programme
- la date de la dernière création du programme compilable associé.

A partir des noms que le catalogue contient et des listes *USE,* on peut constituer le graphe des dépendances entre les unités de l'application. Celui-ci est orienté et sans cycle. Cette dernière propriété est garantie par les commandes gérant le catalogue.

Le nom du catalogue est demandé à l'utilisateur lorsqu'il débute une session.

Pour en changer au cours de la session, il faut utiliser la commande *chcat.*

La commande *pcat* permet d'obtenir la liste des objets contenus dans le catalogue courant. Ces deux commandes sont des appels de procédures écrites en Mentol, ainsi d'ailleurs que toutes les autres commandes que nous décrivons par la suite.

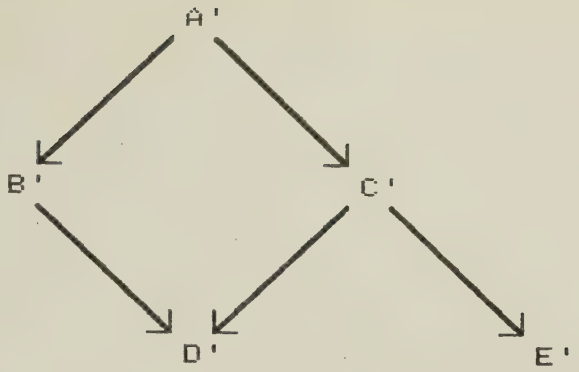
Exemple:

Supposons que le catalogue contienne les programmes A, B, C, D, E et les interfaces A', B', C', D', E' telles que A' (respectivement B', C', D', E') soit une interface sur le programme A (respectivement B, C, D, E).

Supposons que l'on ait les relations de dépendance suivantes entre les interfaces:

B' utilise A'
C' utilise A'
D' utilise C' et B'
E' utilise C'

Alors le graphe représentatif de l'application sera:



Dans cet exemple, une relation telle que A' utilise D' est interdite car elle entraînerait la création d'un cycle dans le graphe des dépendances de l'application. Le pré-processeur vérifie qu'une telle relation n'apparaît pas.

3.2- Définition du contexte de compilation

La commande *use* permet à l'utilisateur d'indiquer la liste des interfaces INTF1, ..., INTFn qui constituent le contexte du programme courant. Cette liste est indispensable pour compiler un programme modulaire.

La commande *use* vérifie l'existence des interfaces citées en examinant le catalogue et ordonne cette liste de la manière suivante:

> Quelle que soit l'interface INTFi apparaissant dans la liste INTF1, ..., INTFi, ..., INTFn, l'interface INTFi ne référence aucune des interfaces INTF1, ..., INTF(i-1).

Elle vérifie également qu'aucun cycle n'est créé dans le graphe de l'application par cette nouvelle relation de dépendance entre les unités de compilation.

Elle insère finalement la liste ordonnée sous forme d'une annotation structurée en tête du programme courant.

3.3- Compilation d'un programme modulaire

La commande *mcompile* permet de compiler un programme modulaire. Pour cela, elle se décompose en trois étapes:

- production d'un programme compilable à partir d'un programme modulaire
- compilation de ce programme
- mise à jour du catalogue.

Un exemple est donné en annexe.

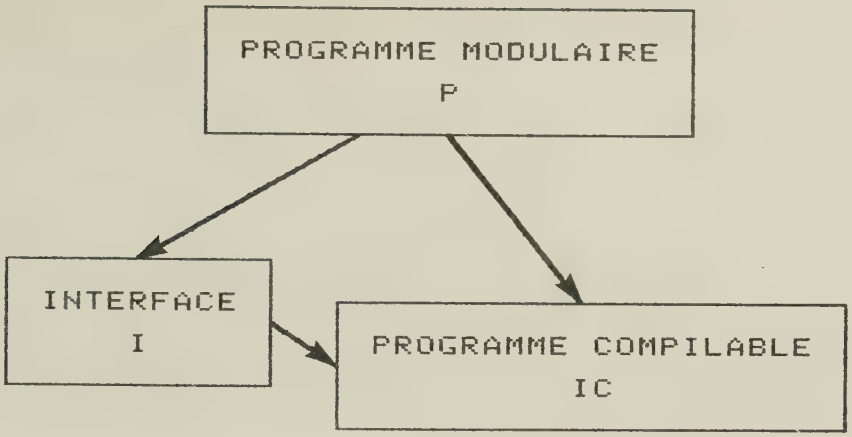
Pour générer le programme compilable, il faut connaître tous les objets importés et établir la liste d'externes correspondant dans le programme compilable. Cela se fait à partir de la liste *USE* du programme modulaire. Il faut aussi connaître les objets exportés afin de pouvoir précéder leur déclaration de l'attribut 'def' dans le programme compilable. Ces objets exportés sont ceux de l'interface associée au programme.

Il est donc nécessaire de construire l'interface sur le programme avant de créer le programme compilable associé. Pour ce faire, deux solutions sont envisageables.

Une solution consiste à exporter systématiquement tous les objets déclarés dans le programme, en particulier toutes les variables et tous les sous-programmes. Cela revient à construire l'interface globale du programme. Cette solution ne tient pas compte des contraintes apportées par le compilateur. En effet, toutes les variables ne sont pas exportables (par exemple les indices de boucles ne le sont pas).

Dans la solution retenue, on exporte sélectivement certains objets. Cette opération se fait à l'aide d'une interface réduite sur le programme, créée interactivement par le programmeur.

Cette phase préliminaire de la commande *mcompile* consiste donc à créer une interface sur le programme. Elle fait appel à la commande *crintf* qui permet de construire une interface réduite. Cette opération est décrite en détails dans la section suivante.



A partir du programme P, de la liste des interfaces qu'il utilise et de l'interface I qui vient d'être construite, le programme compilable IC correspondant peut maintenant être créé. Pour cela, il faut fusionner le programme P et les interfaces INTF1, ..., INTFn (qu'il référence dans la liste *USE* préalablement ordonnée). Cette opération se fait en insérant l'interface INTF1 dans une copie du programme P, puis INTF2 dans le programme résultant de la précédente fusion, puis INTFn. L'insertion se fait zone par zone et elle produit un programme dont les déclarations sont dans un ordre cohérent.

Afin de respecter les mécanismes de compilation séparée, deux modifications doivent être faites sur le programme obtenu. D'une part, on indique que les déclarations que l'on a insérées correspondent à des objets importés en mettant leurs noms dans la liste d'externes du programme créé. D'autre part, les objets laissés dans l'interface I sont exportés, ce qui se fait en ajoutant l'attribut 'def' devant leur déclaration. On obtient ainsi le programme IC qui est alors compilé.

Le nom du programme P et celui de l'interface I sont rangés dans le catalogue, ainsi que la liste *USE* de chacun d'eux et la date à laquelle la compilation a eu lieu.

Si le programme P a déjà été compilé une première fois, la commande *mcompile* fonctionne légèrement différemment. L'utilisateur doit indiquer s'il veut créer une nouvelle interface (dans ce cas, tout ce passe comme précédemment) ou bien réutiliser une ancienne interface sur ce programme. Dans ce dernier cas, il doit donner le nom de cette interface.

S'il ne la modifie pas, le système vérifie la validité de l'ancienne interface sur le nouveau programme. Si l'ancienne interface n'est plus valide, il faut la reconstruire à partir du nouveau programme.

Sinon, si l'interface a été modifiée, le système fournit une liste des composants qui ne sont plus valides, c'est-à-dire les programmes qui utilisent cette interface et qu'il faut donc normalement recompiler, les interfaces réduites construites à partir de l'interface modifiée et éventuellement les programmes et interfaces qui en dépendent.

3.4- Opérations sur les interfaces: réduction et fusion

3.4.1- Réduction d'une interface

Nous décrivons ici la construction d'une interface globale à partir d'un programme modulaire et sa réduction.

L'interface globale d'un programme modulaire est construite de la manière suivante: le corps du programme est supprimé, les zones "label" et "value" sont enlevées, les corps des procédures et des fonctions déclarées au plus haut niveau sont remplacés par la directive 'external'. L'interface globale contient la description de tous les objets déclarés dans le programme.

La cohérence de l'interface globale peut alors être vérifiée. Cette opération revient à vérifier que tous les objets utilisés dans l'interface sont soit entièrement définis dans celle-ci, soit définis dans le contexte de compilation de l'interface. Il suffit en fait de le vérifier pour toutes les constantes et tous les types utilisés dans l'interface.

L'interface globale du programme peut ensuite être réduite. Cette opération se fait interactivement, en conservant la cohérence de l'interface, de la manière suivante.

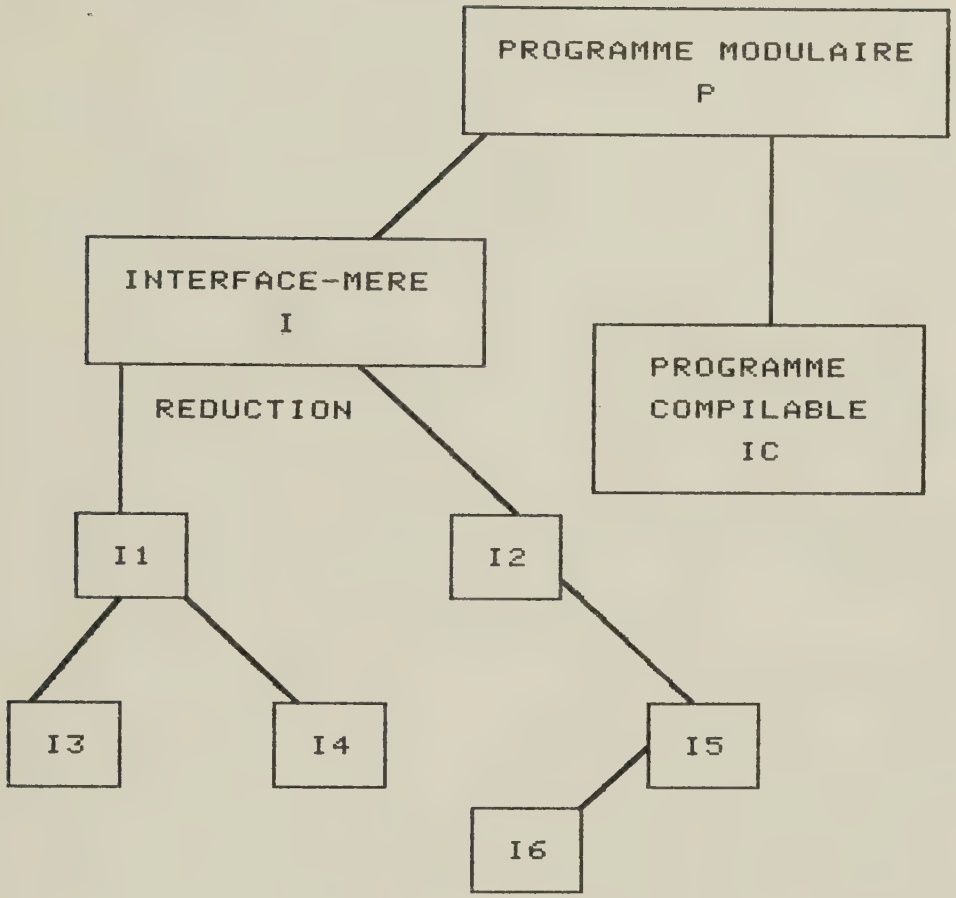
Le système demande à l'utilisateur s'il veut laisser toutes les déclarations de sous-programmes, sinon il examine chaque déclaration et l'enlève si l'utilisateur le désire. Le processus se répète pour les déclarations de variables, puis celles de types et de constantes. Pour les types et les constantes, le système interdit la suppression d'un objet utilisé dans des déclarations déjà conservées dans l'interface.

Une telle réduction est possible soit au cours de la commande *mcompile,* soit à partir d'une interface, par la commande *crintf.*

La commande *crintf* crée une interface à partir d'un programme modulaire ou d'une interface.

Si elle est appliquée à un programme (ce qui est le cas lors de l'appel implicite par la commande *mcompile),* elle crée l'interface globale et la réduit interactivement. Appliquée à une interface qui figure déjà dans le catalogue, elle crée une autre interface par réduction. Cette interface peut alors être utilisée par un autre programme, sans avoir à recompiler le programme dont elle est extraite.

On peut ainsi obtenir un ensemble d'interfaces sur un même programme, par réductions successives, à partir d'une interface-mère créée lors de la commande *mcompile.* Chaque interface dérivée est un sous-ensemble cohérent formé d'objets de l'interface-mère. Toute modification de l'interface-mère invalide les autres interfaces.



La liste *USE* de l'interface réduite peut être calculée à partir de celle de l'interface dont elle est issue. Si une interface n'est pas effectivement utilisée par la nouvelle interface, alors son nom est ôté de la liste.

Finalement, l'utilisateur doit donner un nom à l'interface ainsi créée.

Exemple de création d'interface, extrait d'une session Mentor:

L'interface que l'on veut réduire est la suivante:

```
(*USE*)
program PPILE_INTF;
     const
          MAX     =20;
     type PILE    =
               record
                    P:array[1..MAX]of INTEGER;
                    TOP:INTEGER
               end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function SOMMET(var PIL:PILE):INTEGER;external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

     procedure INIT(var PIL:PILE);external;
     .
```

?.crintf
Voulez-vous laisser tous les sous-programmes? [line]:non

Voulez-vous enlever tous les sous-programmes? [line]:non

Voulez-vous laisser EMPILER ? [line]:oui

Voulez-vous laisser DEPILER ? [line]:oui

Voulez-vous laisser SOMMET ? [line]:non

Voulez-vous laisser PILENONVIDE ? [line]:oui

Voulez-vous laisser INIT ? [line]:non

Voulez-vous laisser tous les types? [line]:non

Voulez-vous enlever tous les types? [line]:oui

Le type PILE est automatiquement conserve

Voulez-vous laisser toutes les constantes? [line]:non

Voulez-vous enlever toutes les constantes? [line]:oui

La constante MAX est automatiquement conservee

Quel nom voulez-vous donner a cette interface?
[ident]:ppile_intf2

L'interface réduite est alors:

```
(*USE*)
program PPILE_INTF2;
     const
          MAX      =20;
     type PILE     =
               record
                    P:array[1..MAX]of INTEGER;
                    TOP:INTEGER
               end;

     procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

     procedure DEPILER(var PIL:PILE);external;

     function PILENONVIDE(var PIL:PILE):BOOLEAN;external;
```

Lors de la réduction de l'interface, la déclaration du type PILE est automatiquement gardée car il est utilisé dans les déclarations des sous-programmes qui sont conservés, de même pour la constante MAX.

La commande *crintf* est ici présentée dans la version destinée à l'apprentissage. Dans la version usuelle, il suffit d'indiquer si l'on veut garder toutes les déclarations ou au contraire toutes les enlever, ou bien de donner une liste des objets que l'on veut conserver.

3.4.2- Fusion de plusieurs interfaces

Une autre méthode pour définir une interface est de créer une interface sur un ensemble d'interfaces. Cette interface peut être soit une réunion de toutes les interfaces considérées, soit une sélection cohérente d'objets de ces interfaces. Elle est obtenue par une fusion des interfaces considérées, suivie éventuellement d'une réduction.

La commande *fusionintf* demande la liste des interfaces que l'on veut englober et le nom de l'interface que l'on va construire. Cette liste est alors ordonnée de la même façon que dans la commande *use* afin d'assurer la cohérence globale de l'interface résultante. Cette interface est obtenue en fusionnant une à une les interfaces de la liste ordonnée, suivant le mécanisme précédemment décrit. La liste *USE* de l'interface résultante est calculée à l'aide des renseignements contenus dans le catalogue. L'interface peut ensuite être réduite interactivement si l'utilisateur ne veut garder que certains objets de chaque interface englobée.

Exemple:

Soient I,J,K trois interfaces que l'on veut fusionner:

```
(*USE*)
program I;
     const
          MAXBUFFERS=16;
     type PRIVATE =@INTEGER;
          BUFFERS =array[1..MAXBUFFERS]of CHAR;
          BUFFSIZES=0..MAXBUFFERS;
          TREES    =PRIVATE;
```

```
            OPERATORS=0..255;
            VARIABLE=PRIVATE;
            LANGUAGE=@INTEGER;
        var ERR:BOOLEAN;
            LANGUE,LOCALANG:LANGUAGE

 (*USE(I)*)
program J;

    function POST(L,T:TREES):TREES;external;

    function HEAD(L:TREES):TREES;external;

    function TAIL(L:TREES):TREES;external;

    function ISEMPTY(L:TREES):BOOLEAN;external;

    function CHILD(N:INTEGER;T:TREES):TREES;external;

    function EQTREE(T1,T2:TREES;L:LANGUAGE):BOOLEAN;external;

 (*USE(I)*)
program K;

    function GETVARIABLE(var NAME:BUFFERS;SIZE:INTEGER):VARIABLE;external;

    procedure SEND(V:VARIABLE;T:TREES);external;

    function RECEIVE(V:VARIABLE):TREES;external;

    function ATOMIDENT(OP:OPERATORS;var NAME:BUFFERS;SIZE:BUFFSIZES):TREES;
     external;

    function MAKEBINARY(OP:OPERATORS;S1,S2:TREES):TREES;external;

    function MAKELIST(OP:OPERATORS;T:TREES):TREES;external;
```

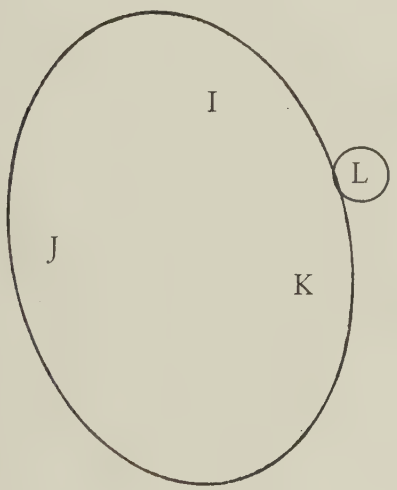
?.fusionintf
Donner la liste des interfaces qu'il faut fusionner:
[lexp]:(i,j,k);
Donner le nom de l'interface ainsi creee:
[ident]:l;
Voulez-vous reduire l'interface globale? (oui/non) [line]:oui
Liste des objets a enlever de l'interface:
[lexp]:;
Liste des objets a laisser dans l'interface:
[lexp]:(head,getvariable);
PRIVATE est automatiquement conserve.
BUFFERS est automatiquement conserve.
TREES est automatiquement conserve.

VARIABLE est automatiquement conserve.
MAXBUFFERS est automatiquement conserve.
lline
L.polish FILE CREATED
L.pascal FILE CREATED
CATALOG.polish FILE OVERWRITTEN
?

L'interface résultant de la fusion est la suivante:

```
    (*USE*)
  program L;
       const
            MAXBUFFERS=16;
       type PRIVATE =@INTEGER;
            BUFFERS =array[1..MAXBUFFERS]of CHAR;
            TREES   =PRIVATE;
            VARIABLE=PRIVATE;

       function HEAD(L:TREES):TREES;external;

       function GETVARIABLE(var NAME:BUFFERS;SIZE:INTEGER):VARIABLE;external;
```

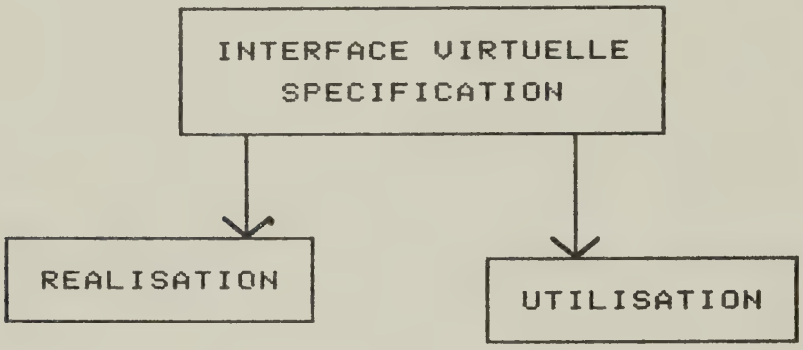
La phase de réduction est présentée ici dans sa version usuelle. Le schéma suivant représente la hiérarchie entre les interfaces I, J, K et L.



3.5- Interface virtuelle

Dans ce qui précède, les interfaces étaient construites à partir de programmes pré-existants. On peut aussi faire l'inverse en commençant par spécifier l'interface.

Une *interface virtuelle* contient des descriptions d'objets qui seront déclarés dans un programme écrit ultérieurement. Les objets décrits dans l'interface sont considérés comme des déclarations du programme associé: il est ainsi inutile de les redéclarer dans le programme. Avec ce mécanisme, il est maintenant possible, en Pascal-Modulaire, de séparer l'en-tête et le corps d'un sous-programme, ce qu'interdit le compilateur Pascal. La séparation entre spécification et réalisation peut donc être effectuée.



Pour réaliser cette opération, il faut conserver l'interface virtuelle IV que l'on vient d'écrire et rentrer son nom dans le catalogue à l'aide de la commande *saveintf*.

Si le programme PR réalise l'interface IV, la commande *realize* appliquée à ce programme demande le nom de l'interface virtuelle qu'il réalise, puis vérifie qu'à chaque description de sous-programme dans l'interface virtuelle correspond bien la déclaration complète dans le programme PR. Elle permet alors d'obtenir un programme modulaire PM dont l'interface effective est identique à l'interface virtuelle IV.

Exemple:

Soit PILEDEFINITION une interface virtuelle:

```
(*USE*)
program PILEDEFINITION;
     const
          N       =28;
     type PILE    =
               record
                    P:array[1..N]of INTEGER;
                    TOP:INTEGER
               end;
```

```
procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

procedure DEPILER(var PIL:PILE);external;

function SOMMET(var PIL:PILE):INTEGER;external;

function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

procedure INIT(var PIL:PILE);external;
```

Voici un programme réalisant cette interface virtuelle:

```
 (*USE(PILEDEFINITION)*)
program PILEREALISATION;

     procedure EMPILER;
          begin
          #
          end;

     procedure DEPILER;
          begin
          #
          end;

     function SOMMET;
          begin
          #
          end;

     function PILENONVIDE;
          begin
          #
          end;

     procedure INIT;
          begin
          #
          end;
```

A partir de l'interface virtuelle et du programme la réalisant, on obtient le résultat suivant par fusion de l'interface avec le programme et en ayant substitué dans chaque déclaration de sous-programmes contenue dans l'interface, 'external' par 'forward'.

La liste *USE* du programme résultant est calculée à partir de celle de l'interface virtuelle et de celle du programme qui la réalise.

```
  (*USE*)
program PILE_R;
     const
        N      =28;
     type PILE    =...;

     procedure EMPILER(#;#);forward;
```

```
procedure DEPILER(#);forward;

function SOMMET(#):INTEGER;forward;

function PILENONVIDE(#):BOOLEAN;forward;

procedure INIT(#);forward;

procedure EMPILER;
     begin
     #
     end;

procedure DEPILER;
     begin
     #
     end;

function SOMMET;
     begin
     #
     end;

function PILENONVIDE;
     begin
     #
     end;

procedure INIT;
     begin
     #
     end;
```

Dans le programme PILEREALISATION réalisant les spécifications décrites dans l'interface PILEDEFINITION, nous ne faisons aucune hypothèse quant à l'ordre dans lequel apparaissent les sous-programmes. Il peut y avoir des références croisées, sans que celles-ci soient indiquées à l'aide de la directive 'forward'. Ce choix a une répercussion sur le programme résultant de la fusion: tous les sous-programmes sont déclarés en deux parties, la première correspond à la spécification, l'autre à la réalisation. La réorganisation des déclarations, qui ne laisserait que les 'forward' indispensables, est une opération coûteuse que nous n'effectuons pas.

Deux méthodologies de programmation sont laissées au choix de l'utilisateur: soit conserver le programme qu'il a écrit, réalisant la spécification, soit le remplacer par celui résultant de la fusion.

Dans le premier cas, pour obtenir un programme compilable, il faut exécuter la commande *realize* appliquée au programme qui réalise l'interface virtuelle. Il faut ensuite appeler la commande *mcompile* pour créer le programme compilable à partir du programme résultant de la fusion du programme source et de l'interface virtuelle. Tous ces programmes ont des noms différents.

L'autre solution revient à oublier le passage par une interface virtuelle et à faire comme si l'interface avait été extraite a posteriori du programme, dès lors que l'on a effectué la commande *realize*. Dans ce cas, on s'éloigne du texte de l'utilisateur, car on le remplace par le programme résultant de la fusion.

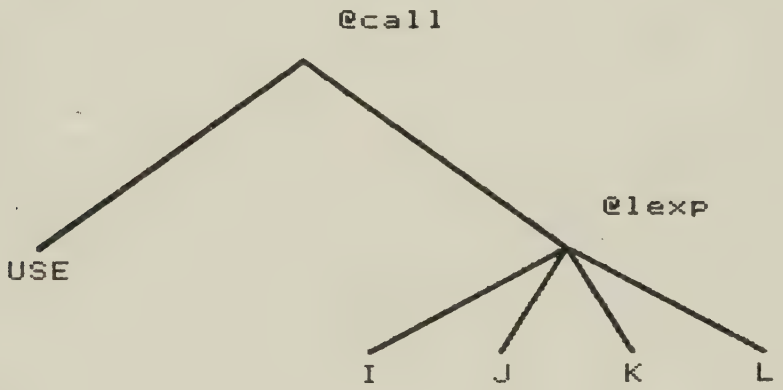
4– Description technique du pré–processeur

Cette section contient une description détaillée du catalogue qui est au coeur du système et des différentes commandes du pré-processeur. Ensuite, certains choix effectués lors de l'implémentation sont développés.

4.1- Le catalogue

4.1.1- La liste USE

Dans un programme modulaire, la liste *USE* est une annotation structurée préfixant l'en-tête du programme. Cette liste est un arbre de type CALL (appel de sous-programme). Le premier fils est l'identificateur USE, le deuxième est une liste d'identificateurs; chaque identificateur est le nom d'une interface.



La liste *USE* est ainsi facilement manipulable sous MENTOR. En particulier, les noms des interfaces qui y apparaissent sont directement accessibles en parcourant l'arbre représentant la liste.

4.1.2- La structure du catalogue

Le catalogue a une structure arborescente manipulable sous Mentor. Il est repéré par une variable Mentol qui est cachée à l'utilisateur. Le nom du catalogue est demandé au début de la session dans l'environnement Pascal Modulaire ou lorsqu'on exécute la commande *chcat* pour changer de catalogue. Il est sauvegardé sous forme arborescente dans le fichier CATALOGUE.polish si son nom est CATALOGUE. La commande *pcat* permet de connaître le contenu du catalogue.

Décompilé à l'aide de la commande *pcat,* le catalogue apparaît comme une liste de noms d'interfaces; à chaque interface sont associés les renseignements suivants:
-la liste *USE* de l'interface
-le nom du programme dont elle est l'interface, s'il existe
-éventuellement, le nom des interfaces qu'elle englobe
-la date de la dernière modification.

```
CP
   04/12/83  15:15
   USE
   sur

CREEDS_INTF
   04/13/83  13:40
   USE(CP)
   sur  CREEDS

VIRT
   04/13/83  11:18
   USE
   sur  VIRTDEF

INTERFACE
   05/03/83  14:45
   USE(CP)
   sur  (VIRT,CREEDS_INTF)
```

Voici la structure abstraite de ce même catalogue:

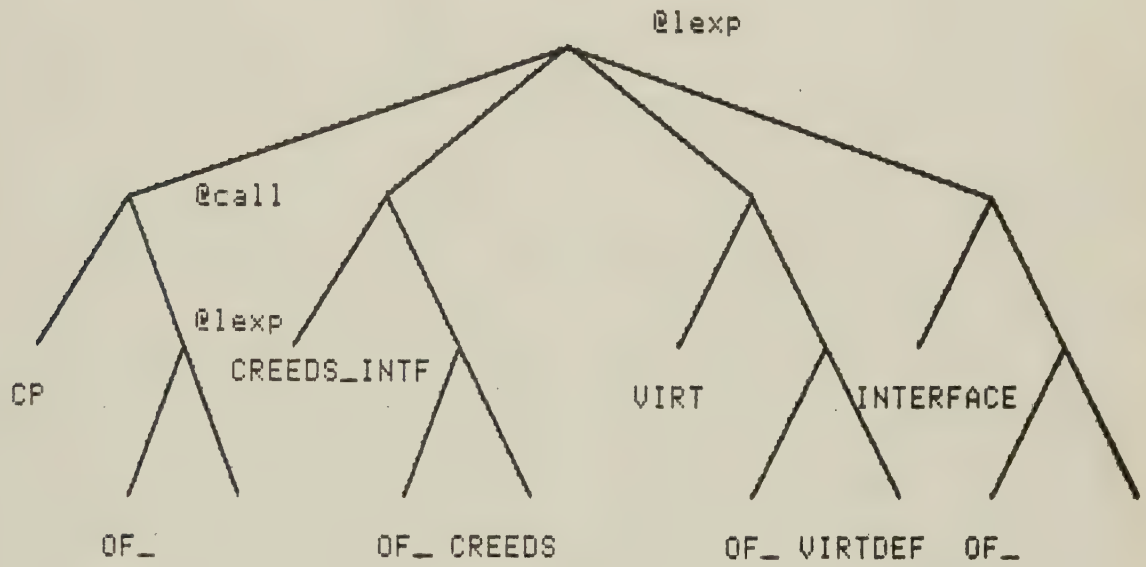
```
lexp
   call
     ident CP
      lexp
        ident OF_
        meta IDENT2
   call
     ident CREEDS_INTF
      lexp
        ident OF_
        ident CREEDS
   call
     ident VIRT
      lexp
        ident OF_
        ident VIRTDEF
```

```
call
  ident INTERFACE
   lexp
     ident OF_
     meta IDENT2
```

L'arbre contenant le catalogue est de type LEXP (liste d'expressions). Chaque expression de la liste est de type CALL et contient tous les renseignements concernant une interface. Le premier fils du noeud CALL est le nom de l'interface. Le deuxième fils est une liste d'expressions se réduisant à deux identificateurs. Le premier contient un identificateur dont le nom n'a aucune importance. Il est noté systématiquement OF_. Le renseignement important est contenu dans une annotation structurée qui lui est associée. Elle n'existe que dans le cas où l'interface englobe plusieurs interfaces. Cette annotation contient alors la liste des noms des interfaces considérées. C'est une liste d'identificateurs.

Le deuxième identificateur est le nom du programme associé à l'interface, s'il existe. S'il n'existe pas, l'interface est soit une interface virtuelle, soit une interface qui englobe d'autres interfaces.

Voici l'arborescence correspondant au catalogue précédent:



Les annotations structurées associées à certains noeuds de l'arbre n'apparaissent pas ci-dessus.

Pour chaque élément de la liste, c'est-à-dire pour chaque noeud de type CALL, on associe deux annotations: l'une est préfixée et contient la date de dernière modification, l'autre est postfixée et contient la liste *USE* de l'interface.

Dans le cas où un nom de programme est associé à celui de l'interface, deux autres annotations sont placées sur le noeud de type LEXP. L'annotation préfixée contient la date et celle postfixée contient la liste *USE* du programme.

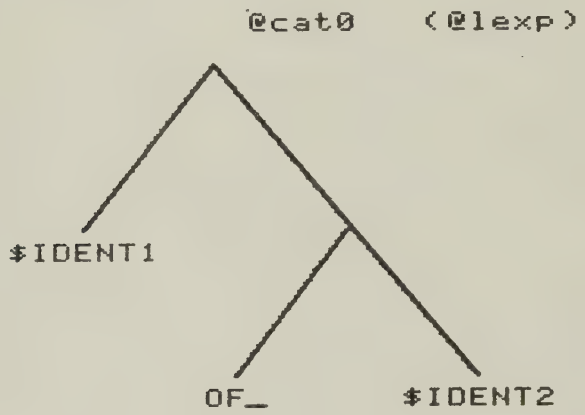
Les listes *USE* apparaissant dans le catalogue ont la même structure que dans l'en-tête du programme ou de l'interface qu'elles annotent.

Le catalogue contient donc les noms des programmes et de leurs interfaces. Il ne contient pas le texte des composants eux-mêmes.

L'examen du catalogue permet d'ordonner les listes *USE* des programmes et des interfaces, de connaître les composants invalidés après modification d'un programme.

Les opérations effectuées sur le catalogue sont:
-des insertions de noms d'interfaces ou de programmes
-des modifications concernant les listes *USE* des interfaces ou des programmes et les mises à jour des dates
-des recherches de noms afin de trier les composants de l'application.

L'insertion d'un nouveau composant dans le catalogue se fait de la manière suivante. On insère à la fin du catalogue le shéma prédéfini suivant:



Lorsqu'on catalogue un nouveau composant (programme ou interface), on commence par ranger le nom de l'interface qui vient d'être créée. Pour cela on remplace la méta-variable $IDENT1 par le nom de l'interface. La méta-variable $IDENT2 contiendra ensuite le nom du programme associé à l'interface. Les annotations préfixée et postfixée sont obtenues par copie des annotations contenues dans l'interface (date et liste *USE).*

Dans le catalogue, les interfaces ont un rôle plus important que les programmes. En effet, ce sont elles qui apparaissent dans les listes *USE.* Il est plus aisé d'avoir directement, dans le catalogue, les noms des interfaces pour ordonner cette liste. Il est ensuite facile d'obtenir le nom de toutes les interfaces construites sur un programme, par calcul sur le catalogue (la commande *lintf* le fait).

Le catalogue, parcouru de gauche à droite (au sens parcours d'arbre), donne directement les liens de dépendance entre les composants. En effet, lorsqu'un nouveau composant est créé, son nom est inséré à droite dans l'arbre représentant le catalogue. Un programme ou une interface ne peut utiliser que des interfaces figurant déjà dans le catalogue, donc situées

à gauche dans le catalogue.

Lorsqu'un composant est modifié et si son contexte de compilation reste le même, l'ordre du catalogue est inchangé. Par contre, si la liste *USE* est modifiée, il faut réordonner le catalogue afin de conserver la propriété suivante:

> un programme ou une interface figurant dans le catalogue ne peut utiliser que des interfaces dont le nom est situé à gauche dans le catalogue.

Le respect de cette propriété interdit la création de cycles dans le graphe de dépendances entre les composants de l'application.

Cette propriété définit un ordre partiel entre les interfaces (ou les programmes associés): soient I et J deux interfaces, I est à gauche de J dans le catalogue est équivalent à dire que I ne référence aucun objet déclaré dans J.

Elle permet d'effectuer le tri du catalogue. La commande *tri* donne un ordre possible entre les composants du catalogue. Dans le cas où le graphe des dépendances contient un cycle, la commande échoue. Il n'y a bien sûr pas unicité de l'ordre donné.

4.2- Description détaillée des commandes du pré-processeur

Cette section contient une description technique du pré-processeur. Une explication détaillée de chaque commande est donnée et les informations nécessaires pour l'exécuter sont fournies.

Les commandes qui ont été partiellement décrites dans la troisième partie sont les commandes principales. Elles enchaînent des commandes élémentaires. Celles-ci sont décrites dans ce qui suit. Leur mode d'emploi n'est pas donné, car elles ne sont utilisées que par appel implicite. Dans le titre des sections qui les décrivent, leur nom est suivi de '+'.

4.2.1- La commande use

La commande *use* permet de définir le contexte de compilation du programme courant. Cette commande interactive demande à l'utilisateur une liste d'identificateurs qui sont les noms des interfaces utilisées par le programme.

Elle effectue le tri de la liste d'interfaces. Ce tri est réalisé en lisant le catalogue de droite à gauche, en gardant les noms d'interfaces apparaissant dans la liste. Si un nom de la liste ne figure pas dans le catalogue, il n'est donc pas conservé dans la liste résultante.

Le noeud de type CALL est alors construit. Son premier fils est l'identificateur USE, le deuxième est la liste triée. Ce noeud est placé dans une annotation structurée préfixant l'en-tête du programme. Il est alors repéré par la variable @k 1 b0, le programme étant repéré par la variable @k.

Si la liste *USE* d'un programme ou d'une interface est omise, on ne considère pas que le contexte de compilation est vide, mais qu'il n'a pas été indiqué. Si on y applique ensuite une commande nécessitant la connaissance du contexte de compilation, elle échouera.

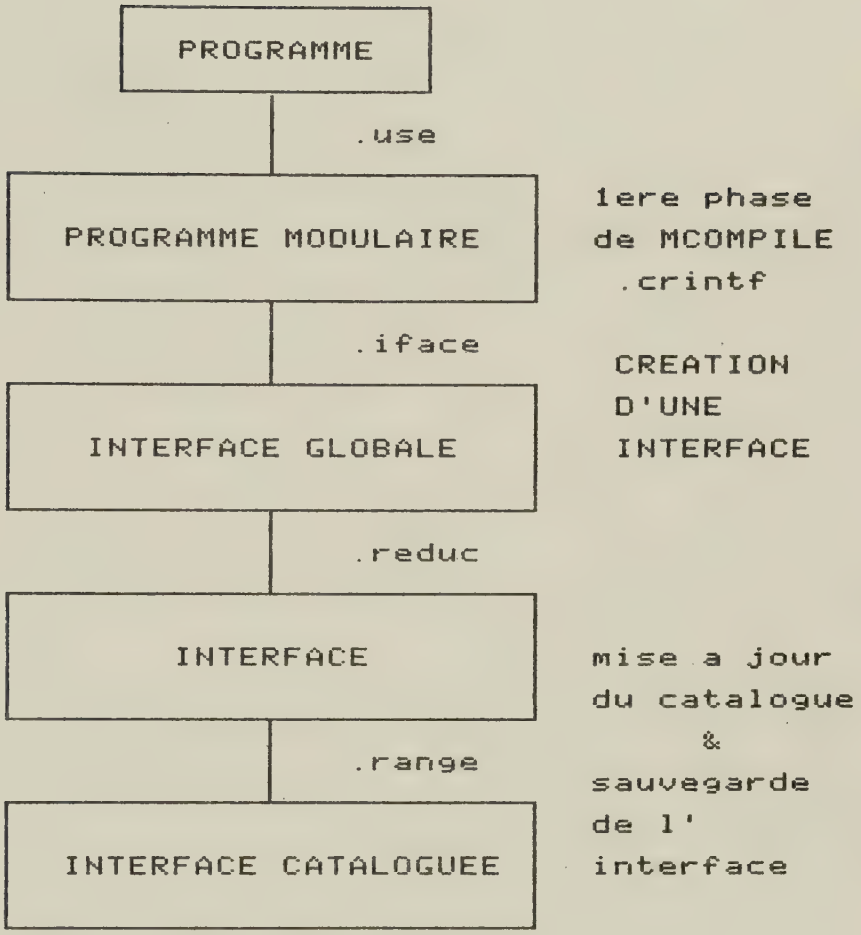
4.2.2- La commande mcompile

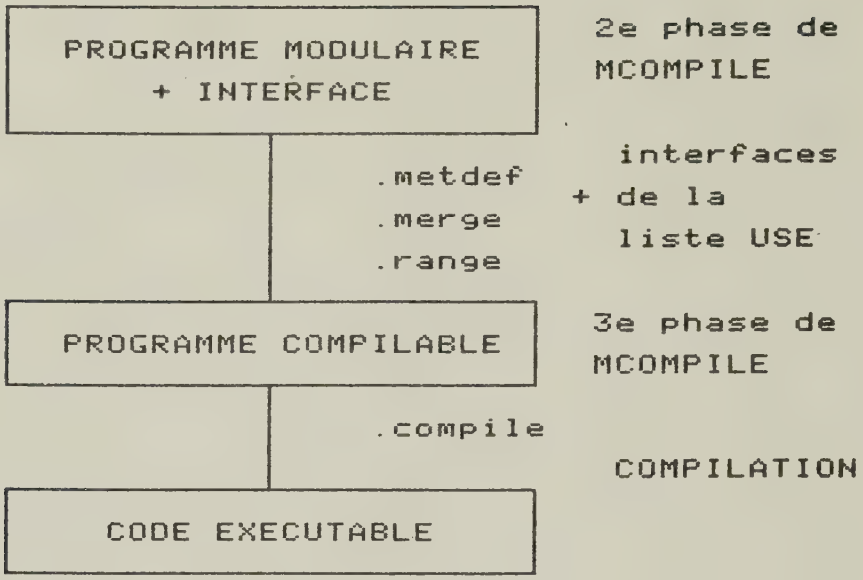
Cette commande s'applique sur le programme courant (désigné par @k). Ce programme modulaire doit avoir une liste *USE* (éventuellement vide), obtenue par la commande *use*. Sinon la commande est interrompue.

Elle appelle implicitement les commandes suivantes: *iface, reduc, range, metdef, merge, range, compile*. Les descriptions de chaque commande sont contenues dans les sections suivantes.
Le schéma suivant donne l'enchaînement des commandes.

Lorsque l'exécution est terminée, le programme courant est inchangé. L'interface et le programme compilable sont repérés par les variables @interface et @compilable.

Le nom du programme compilable est créé automatiquement à partir de celui de l'interface en y ajoutant la lettre 'C'.


PROGRAMME
.use
PROGRAMME MODULAIRE
1ere phase
de MCOMPILE
.crintf
.iface
CREATION
D'UNE
INTERFACE
INTERFACE GLOBALE
.reduc
INTERFACE
mise a jour
du catalogue
&
.range
sauvegarde
de l'
INTERFACE CATALOGUEE
interface



4.2.3- La commande crintf

La variable courante désigne un programme (ou une interface) sur lequel on veut créer une interface réduite. Appliquée à un programme, elle fait d'abord appel à la commande *iface*. Elle appelle la commande *reduc* et ensuite met à jour le catalogue. Pour cela, elle range le nom de la nouvelle interface à la fin du catalogue et lui associe le nom du programme ou de l'interface à partir duquel elle a été obtenue. La date de création et la liste *USE* sont insérées dans le catalogue.

4.2.4- La commande iface +

La commande *iface* crée l'interface globale du programme courant. Elle supprime le corps du programme et les zones "label" et "value". Dans les sous-programmes déclarés au niveau global, elle remplace leur corps par la directive 'external'. Dans le cas où un sous-programme est déclaré en deux parties à l'aide de la directive 'forward', elle enlève la déclaration complète et remplace dans son en-tête 'forward' par 'external'.

4.2.5- La commande reduc +

La commande *reduc* permet la réduction interactive d'une interface, en respectant sa cohérence.

Il existe deux présentations de cette commande. La première guide pas-à-pas l'utilisateur. La réduction se fait dans l'ordre inverse des déclarations, afin de ne pas enlever un objet utilisé dans une déclaration déjà conservée. Elle demande pour chaque objet déclaré dans l'interface s'il veut ou non le laisser figurer dans l'interface réduite. Si cet objet n'est pas utilisé dans la suite des déclarations, il est enlevé.

Dans la seconde présentation qui est la version par défaut, l'utilisateur donne une liste d'objets à enlever ou à laisser dans l'interface.

Le système demande d'abord une liste des objets qui ne figureront plus dans l'interface réduite. La suppression d'un objet utilisé dans les déclarations conservées est interdite. Cette vérification est faite automatiquement par le système. Si la réponse à cette question est une liste vide, le système demande alors de donner une liste des objets qu'il faut conserver. Les objets nécessaires pour respecter la cohérence sont alors rajoutés automatiquement. Si cette dernière réponse est encore une liste vide, l'interface obtenue ne contient aucune déclaration.

4.2.6- La commande metdef +

Cette commande est appelée implicitement par *mcompile,* lorsque l'interface du programme courant vient d'être construite ou, dans le cas d'une recompilation, lorsqu'elle vient d'être chargée dans la variable @interface.

Pour chaque variable et chaque sous-programme décrits dans l'interface, elle indique que ces objets sont exportés en mettant l'attribut 'def' devant leur nom dans le programme compilable.

Dans le cas où on recompile le programme sans modifier son interface, elle vérifie que cette interface reste valide sur le programme, c'est-à-dire qu'elle peut être obtenue à partir du programme par les commandes *iface* et *reduc.* En particulier si une variable ou un sous-programme figure dans l'interface mais pas dans le programme, il ne sera pas possible de mettre 'def' devant son nom, l'exécution s'arrête sur une erreur: l'interface n'est pas valide. Si ces vérifications échouent, l'interface doit être redéfinie à partir du programme courant.

4.2.7- La commande merge +

Cette commande est appelée par *mcompile* pour effectuer la fusion structurée des interfaces utilisées et du programme courant afin d'obtenir un programme Pascal standard. Pour chaque interface de la liste *USE* ordonnée, la fusion se fait zone par zone. Les déclarations de sous-programmes apparaissant dans les interfaces sont insérées sans modification puisqu'elles

contenaient déjà la directive 'external' dans les interfaces. Afin de respecter les mécanismes de compilation séparée, les noms des variables et des sous-programmes importés sont placés dans la liste d'externes du programme résultant de la fusion.

Le programme ainsi créé est compilé.

4.2.8- La commande saveintf

Cette commande sauvegarde une interface virtuelle et met à jour le catalogue, en y rangeant son nom, sa liste *USE* et sa date de création. Elle vérifie que le nom de l'interface n'existait pas auparavant dans le catalogue.

4.2.9- La commande ecrit

Cette commande s'applique sur un programme ou une interface qui a été modifié et dont le nom est déjà dans le catalogue. Elle sauvegarde le programme ou l'interface et met à jour le catalogue.

4.2.10- La commande fusionintf

Cette commande permet de construire une interface englobant des interfaces pré-existantes. Elle demande à l'utilisateur la liste des interfaces qu'il faut fusionner, elle l'ordonne et effectue ensuite la fusion de ces interfaces.

Elle demande à l'utilisateur s'il veut réduire l'interface ainsi obtenue, si oui elle appelle *reduc*.

La liste *USE* de l'interface résultante est alors calculée.

L'utilisateur doit finalement donner un nom à l'interface qui est sauvegardée. Ce nom est rangé dans le catalogue, ainsi que sa liste *USE* et sa date de création. Dans le sous-arbre du catalogue correspondant à cette interface, l'identificateur OF_ est annoté par la liste des interfaces englobées.

La variable courante contient finalement l'interface résultante.

Cette commande échoue si l'une des interfaces n'a pas de liste *USE* ou si la liste des interfaces qu'il faut fusionner se réduit à une interface.

4.2.11- La commande realize

Le programme courant réalise les spécifications qui sont décrites dans une interface virtuelle. Le nom de cette interface doit déjà figurer dans le catalogue. La commande *realize* crée un programme modulaire à partir de ce programme et de l'interface virtuelle. Pour cela, elle s'applique sur le programme source et demande le nom de l'interface virtuelle qu'il réalise. Le programme modulaire est obtenu par insertion de l'interface dans une copie du programme source, après avoir remplacé 'external' par 'forward' dans chaque déclaration de

sous-programme contenue dans l'interface. Le nom de ce programme est demandé à l'utilisateur. Si le nom donné est distinct ou non de celui du programme source, on retrouve la première ou la seconde méthodologie décrite dans la section 3.5.

Dans une interface virtuelle, il n'est pas possible de mélanger des concepts de modularité et des mécanismes de compilation séparée traditionnelle. Tous les en-têtes de sous-programmes de l'interface sont des spécifications qui sont réalisées dans le programme associé. Ils ne peuvent pas être des sous-programmes externes compilés séparément.

La commande *realize* vérifie que l'interface virtuelle est une interface valide sur le programme. Pour les constantes, les types et les variables, il n'y a rien à vérifier. Par contre, il faut nécessairement qu'à chaque en-tête de sous-programme dans l'interface corresponde une déclaration complète dans le programme. Les paramètres du sous-programme ne doivent pas être dupliqués dans le programme.

Cette commande associe le nom du programme à celui de l'interface dans le catalogue. La liste *USE* et la date de création sont également rangées dans le catalogue.

4.2.12- La commande checkintf

Cette commande s'applique sur une interface dont on veut vérifier la cohérence. Il est inutile de vérifier la cohérence d'une interface réduite si l'on a préalablement vérifié celle de l'interface dont elle est issue, car la réduction n'entraîne pas d'incohérence.

A partir du contexte de compilation de l'interface, elle construit l'ensemble des objets utilisables dans l'interface. Cet ensemble se réduit aux constantes et aux types qui sont définis dans l'interface elle-même et dans les interfaces citées dans sa liste *USE*. En effet, les variables et les sous-programmes ne peuvent pas apparaître dans la déclaration d'une constante, d'un type, d'une autre variable ou d'un autre sous-programme. Il est donc inutile d'en tenir compte pour cette vérification.

La commande vérifie ensuite que chaque constante et chaque type utilisé dans l'interface appartient au contexte de compilation. Il faut examiner toutes les parties droites des déclarations.

Pour les déclarations de constantes, si la partie droite contient un identificateur, il doit appartenir au contexte: c'est une constante définie dans une des interfaces utilisées. Sinon c'est une constante de type entier, réel ou caractère.

Il faut également examiner chaque sous-arbre correspondant à une partie droite de déclaration de type et de variable et tester si les identificateurs qui y sont contenus appartiennent au contexte de compilation.

Pour les déclarations de sous-programmes, il faut examiner les types des paramètres.

Cette commande, ainsi que celle qui est décrite dans le paragraphe suivant, est une opération coûteuse qui est faite à la demande. Elle pourrait être appelée par la commande *mcompile* juste après la création de l'interface globale.

4.2.13- La commande checkuseintf

Cette commande s'applique sur une interface. Elle en vérifie la cohérence et réduit sa liste *USE* si certaines interfaces ne sont pas effectivement utilisées. Elle exécute *checkintf* et pour chaque interface de la liste, elle vérifie qu'au moins un objet (constante ou type) est référencé dans l'interface, sinon le nom de l'interface est enlevé de la liste *USE*.

Cette commande est coûteuse car de nombreuses vérifications sont faites. Par exemple, il faut tenir compte des situations du type suivant:
soient A, B deux interfaces telles que B utilise A et P un programme qui utilise A et B et contenant les déclarations suivantes

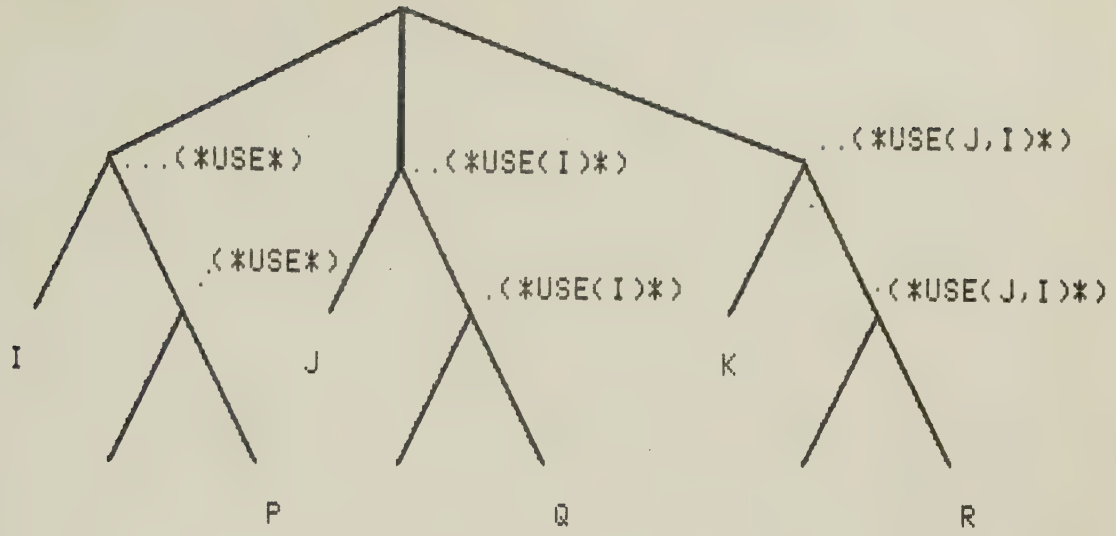
```
(*USE*)
program A;
type T=...;
...

(*USE(A)*)
program B;
type R=array... of T;
...

(*USE(B,A)*)
program P;
var X:R;
...
```

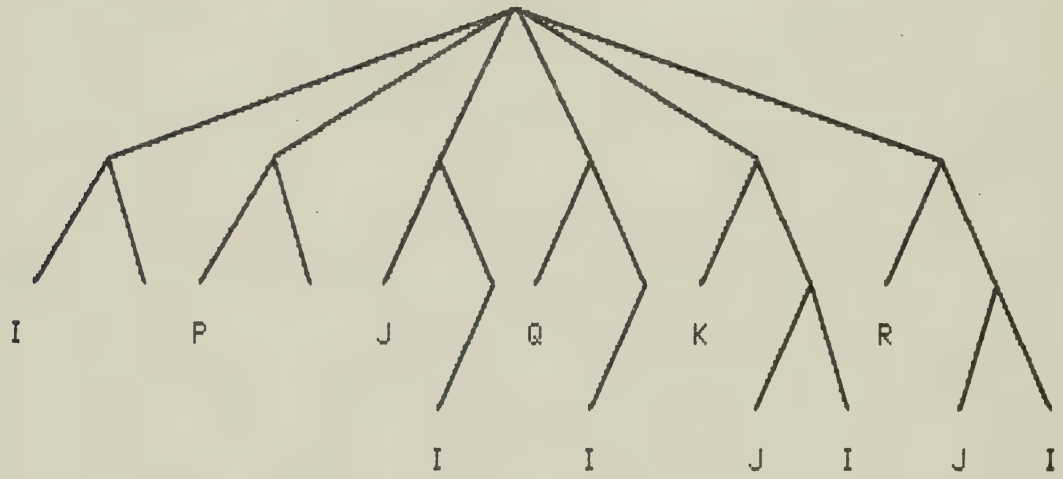
Le programme P utilise indirectement l'interface A. La création du programme compilable à partir de P se fera en insérant la déclaration du type R contenu dans B. Pour obtenir un programme correct, il faudra également insérer la déclaration du type T utilisé dans la définition de R. Les interfaces utilisées par P sont effectivement A et B.

4.2.14- La commande querecomp +

Lors de la recompilation d'un programme, si son interface a été modifiée, les programmes qui l'utilisent ne sont plus valides et doivent être recompilés. La commande *querecomp,* appliquée à une variable désignant le nom d'une interface, donne une liste ordonnée des interfaces et programmes invalidés après la modification de cette interface. Dans le catalogue, elle indique que ces composants sont invalidés.

Elle transforme le catalogue en une liste d'expressions (désignée par la variable @gus). Chaque expression est de type CALL. Le premier fils est le nom d'une interface ou d'un programme; le deuxième est la liste des interfaces utilisées.





Elle fait ensuite appel au programme Pascal ORDON_IC. Ce programme est associé au programme modulaire ORDON (cf Annexes), qui utilise l'interface TREEFRONT_I qui permet de manipuler des variables et des arbres Mentol en Pascal.

Le programme ORDON crée le graphe des dépendances entre les programmes et les interfaces à partir des données contenues dans @gus. Il va ensuite trouver tous les successeurs de l'interface modifiée, c'est-à-dire tous les programmes et toutes les interfaces qui l'utilisent, et les ordonner. Pour cela, il va chercher tous les chemins du graphe dont le point de départ est le composant modifié. Il ordonne ensuite les points par niveaux. Le niveau 0 est celui du composant modifié. Le niveau 1 correspond à ses successeurs directs dans le graphe. La liste ordonnée ainsi obtenue se trouve alors dans la variable Mentol @arecomp. Les programmes dont le nom est contenu dans la liste doivent être recompilés pour remettre l'application dans un état cohérent.

Dans le catalogue, la date associée à chaque composant invalidé est remplacée par "invalide". Ce commentaire sera remplacé par la date de la recompilation pour un programme ou par la date de la remise à jour pour une interface.

La commande *querecomp* appliquée à l'interface I du catalogue précédent donne pour résultat la liste suivante:

(J,Q,K,R)

Il faudra recompiler le programme Q puis le programme R, en utilisant les interfaces J et K respectivement.

4.3- Remarques diverses

4.3.1- Sauvegarde des programmes et des interfaces

Le catalogue précédemment décrit ne contient pas le texte des programmes et interfaces. Par défaut, ceux-ci sont sauvegardés seulement sous forme arborescente, lors des commandes *mcompile, crintf, fusionintf, saveintf, ecrit, realize.* La commande *listing* effectuée en début de session ou avant l'exécution de l'une de ces commandes permet de sauvegarder les composants créés sous forme arborescente et sous forme de texte. Elle permet également de conserver le programme compilable sous les deux formes. Par défaut, ce programme n'est gardé sous aucune forme.

4.3.2- Choix des noms des interfaces et des programmes compilables

Lors de l'écriture d'une interface virtuelle, l'utilisateur choisit lui-même le nom qu'il veut lui donner. Cela se passe de même pour les interfaces construites à partir d'un programme.

Par contre, le nom du programme compilable est construit automatiquement à partir de celui de l'interface associée en y ajoutant la lettre 'C'. Il existe ainsi une relation simple entre le nom de l'interface qui contient la description des objets exportés et le nom du segment qui contient les points d'entrée correspondants.

4.3.3- Le traitement des messages

Les messages apparaissant lors du déroulement interactif de certaines commandes du pré-processeur sont manipulés de façon uniforme à l'aide de la commande *prmess.* Les messages sont contenus dans le fichier MODULAIRE.messages (cf Annexe).

Un message est référencé par le nom du module qui le contient et un numéro qui est attaché à ce message dans le module. [LAN 83]

4.3.4- La documentation du pré-processeur

Une documentation sur le pré-processeur est disponible lors d'une session Mentor dans l'environnement Pascal Modulaire.

La commande *liste* donne une liste des commandes du pré-processeur, pour lesquelles il existe des informations.

La commande *detail* demande le nom d'une commande et fournit les informations nécessaires pour l'exécuter.

4.3.5- Problème lié au compilateur Pascal-Multics

Le compilateur Pascal Multics impose des restrictions aux extensions présentées dans la première partie. En effet, il ne permet pas d'exporter à partir de deux programmes distincts des variables ou des sous-programmes ayant le même nom. Il n'est donc pas possible d'associer plusieurs réalisations à une même spécification.

D'autre part, si l'on définit sur un même programme deux interfaces ayant des objets (variables ou sous-programmes) en commun, la compilation du programme compilable associé à la première interface se fera sans problème. Par contre, la compilation du second sera accompagnée d'un message indiquant que les objets communs aux deux interfaces ne peuvent être exportés par ce second programme (ils sont déjà exportés par le premier programme).

Ce problème peut être résolu pour les sous-programmes en les déclarant de la façon suivante dans le programme qui les importe

```
procedure P; external 'IC$p';
```

où IC est le nom du programme qui exporte la procédure P.

La commande *intfbind* permet de transformer une interface I en une interface où tous les en-têtes de sous-programmes P sont suivis de:

```
'external IC$p'
```

(IC est le nom du programme compilable associé à l'interface I)

Il faut ensuite utiliser la commande *bind* qui appelle la commande Multics du même nom.

Pour l'exportation des variables, il n'y a pas de solution permettant de résoudre le problème.

4.3.6- Problèmes posés par les fichiers

Le compilateur Pascal-Multics impose la déclaration des fichiers INPUT et OUTPUT dans l'en-tête du programme, si ces fichiers sont utilisés dans le programme par l'intermédiaire des procédures d'entrée-sortie (read, readln, write et writeln). En particulier, si ces fichiers sont

référencés implicitement par un sous-programme importé, il faut également compléter l'en-tête du programme.

Dans le pré-processeur, l'en-tête du programme compilable n'est pas complété automatiquement s'il utilise une interface référençant des procédures d'entrée-sortie. L'utilisateur doit faire lui-même les ajouts nécessaires.

D'autre part, la réduction d'une interface ne modifie pas son en-tête: si le seul sous-programme appelant une procédure d'entrée-sortie ne figure plus dans l'interface réduite, alors l'en-tête contiendra toujours input ou output. Aucune vérification sémantique n'est faite. Ajouter systématiquement à l'en-tête du programme compilable ceux des interfaces utilisées n'est pas une solution acceptable. Elle peut entraîner des déclarations superflues et non souhaitées.

Un autre problème est posé avec la déclaration des fichiers. Le compilateur Pascal n'autorise pas la déclaration d'un fichier global dans un programme non principal. Le transport des déclarations des fichiers dans le programme principal n'est pas effectué automatiquement comme cela se fait dans Adele [EST 82].

4.3.7- Un Pascal Modulaire avec Métal

On pourrait définir un nouveau formalisme sous Mentor décrivant le Pascal Modulaire et non plus le Pascal Multics. Pour cela, il faudrait étendre le Pascal standard en y ajoutant les points suivants:

- rajouter une zone USE dans la liste des déclarations. Un programme modulaire se présenterait ainsi:

```
program P;
use A,B,C;
const ...
.
```

- pour distinguer un programme et une interface, rajouter le mot réservé INTERFACE et une zone ON indiquant sur quoi porte l'interface. Une interface se présenterait ainsi:

```
interface I;
on P;
use A,B,C;
const ...
.
```

- toujours pour les interfaces, autoriser que le corps d'une procédure ou d'une fonction soit vide. On aurait ainsi:

```
interface I;
on P;
use A,B,C;
```

```
procedure PROC(var X:INTEGER);
function F:INTEGER;
.
```

- lors de l'écriture d'une interface virtuelle, pour faire la distinction entre un sous-programme externe au sens normal de la compilation séparée et un sous-programme dont le corps est défini ultérieurement dans le programme qui réalise cette spécification, introduire une nouvelle directive XFORWARD. Le corps d'un sous-programme externe contiendra toujours la directive EXTERNAL. Par contre, la déclaration d'un sous-programme en deux parties (en-tête et corps séparés dans deux programmes) se fera à l'aide de XFORWARD. Cette déclaration est analogue à celle d'un sous-programme en deux parties dans un même programme, à l'aide de FORWARD.

Nous avons préféré utiliser Mentor adapté au langage Pascal standard. Ce choix nous a permis de nous servir des outils déjà développés dans l'environnement Pascal. De plus, ceci permet à l'utilisateur de manipuler des programmes Pascal modulaire et des programmes Pascal Multics dans le même environnement Mentor.

Conclusion

Les mécanismes présentés respectent la structure du Pascal standard auquel ils apportent très peu de modifications syntaxiques ou sémantiques. Ils permettent de garantir la vérification de cohérence pour toute une application (sans nécessiter une description multiple des objets partagés, source d'erreurs) et de mieux maîtriser sa structure par la connaissance des dépendances entre composants.

Ces mécanismes peuvent être facilement inclus dans un compilateur ou traité par un pré-processeur comme nous l'avons nous-même fait en utilisant des outils (compilateur, Mentor) appartenant à l'environnement Pascal standard sous Multics.

Par ailleurs, ces mécanismes ne sont pas coercitifs et aucune méthodologie de programmation n'est, a priori, sous-jacente. Ainsi, rien n'oblige à structurer une application de manière arborescente ou à effectuer une décomposition systématique entre spécification et réalisation même si ces méthodologies sont les plus utilisées. Ils offrent donc une grande souplesse au programmeur pour le découpage de son application et la construction des programmes.

Le pré-processeur a été réalisé à l'aide de l'environnement de programmation Mentor pour le langage Pascal. Il transforme des programmes Pascal-modulaire en programmes compilables.

En vue du transport d'une application sur un autre compilateur, il est possible de restructurer une application écrite en Pascal standard et de transformer les programmes Pascal en programmes Pascal-modulaire. Des conditions sont cependant requises. Tout d'abord l'existence de Mentor pour manipuler les nouveaux programmes est imposée. Ensuite, il faut adapter le pré-processeur pour qu'il construise des programmes acceptables par le nouveau compilateur. Pour cela, il suffit de changer la manière d'indiquer qu'un objet est exporté ou importé. Le pré-processeur n'étant pas inclus dans le compilateur, il est donc facilement portable.

Le pré-processeur est actuellement utilisable sous Multics. Une étape ultérieure de son développement sera sa traduction en Pascal afin d'améliorer son efficacité. Ce travail se fera en utilisant le Pascal modulaire, ce qui va permettre de tester son efficacité sur une application réelle (il y a environ deux mille lignes de programmes Mentol).

Annexes

L'annexe contient:
- une session Mentor dans l'environnement Pascal Modulaire
- les fichiers Mentol et le fichier MODULAIRE.messages
- les programmes Pascal TREEFRONT_I et ORDON

1– Session dans l'environnement Pascal Modulaire sous Mentor

Les commentaires sur le déroulement de la session sont rajoutés en italiques dans le texte. L'étape d'édition des programmes sous Mentor n'est pas décrite.

```
mentor
- MENTOR  VERSION 4.1 - JAN-82  ON MULTICS (IREP)
User name:valerie
VALERIE.prelude FILE LOADING
Donner le nom du catalogue dans lequel vous voulez travailler:
[ident]:catalogue
Le catalogue est vide.
?
```

Voici le programme modulaire que l'on veut compiler. Sa liste USE doit être vide car c'est le premier programme de l'application.

```
?.use
[lexp]:;
?p*

(*USE*)
program PPILE;
      const
            MAX=15;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:0..MAX
                  end;

      procedure EMPILER(var PIL:PILE;ITEM:INTEGER);
            begin
```

```
        if PIL.TOP<MAX then
            begin
            PIL.TOP:=PIL.TOP+1;
            PIL.P[PIL.TOP]:=ITEM
            end;
        end;

    procedure DEPILER(var PIL:PILE);
        begin
        if PIL.TOP#0 then PIL.TOP:=PIL.TOP-1;
        end;

    function SOMMET(var PIL:PILE):INTEGER;
        begin
        if PIL.TOP#0 then SOMMET:=PIL.P[PIL.TOP]
        else SOMMET:=-1234567890;
        end;

    function PILENONVIDE(var PIL:PILE):BOOLEAN;
        begin
        if PIL.TOP=0 then PILENONVIDE:=FALSE
        else PILENONVIDE:=TRUE;
        end;

    procedure INIT(var PIL:PILE);
        begin
        PIL.TOP:=0;
        end;

?.mcompile
lline
PPILE.polish   FILE CREATED
Voulez-vous reduire l'interface globale? (oui/non) [line]:non
Quel nom voulez-vous donner a l'interface? [ident]:ppileintf
lline
PPILEINTF.polish   FILE CREATED
CATALOGUE.polish   FILE CREATED
lline
PPILEINTFC.polish   FILE CREATED
PPILEINTFC.pascal   FILE CREATED
PASCAL 6.02
?@interface p*
```

L'interface du programme est désignée par la variable @interface.

```
(*USE*)
program PPILEINTF;
      const
          MAX=15;
      type PILE=
                record
                     P:array[1..MAX]of INTEGER;
                     TOP:0..MAX
                end;

      procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

      procedure DEPILER(var PIL:PILE);external;

      function SOMMET(var PIL:PILE):INTEGER;external;

      function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

      procedure INIT(var PIL:PILE);external;

?@compilable p*
```

Le programme compilable est désigné par la variable @compilable.

```
program def PPILEINTFC;
      const
          MAX=15;
      type PILE=
                record
                     P:array[1..MAX]of INTEGER;
                     TOP:0..MAX
                end;

      procedure def EMPILER(var PIL:PILE;ITEM:INTEGER);
          begin
          if PIL.TOP<MAX then
                begin
                PIL.TOP:=PIL.TOP+1;
                PIL.P[PIL.TOP]:=ITEM
                end;
          end;

      procedure def DEPILER(var PIL:PILE);
          begin
          if PIL.TOP#0 then PIL.TOP:=PIL.TOP-1;
          end;
```

```
        function def SOMMET(var PIL:PILE):INTEGER;
                begin
                if PIL.TOP#0 then SOMMET:=PIL.P[PIL.TOP]
                else SOMMET:=-1234567890;
                end;

        function def PILENONVIDE(var PIL:PILE):BOOLEAN;
                begin
                if PIL.TOP=0 then PILENONVIDE:=FALSE
                else PILENONVIDE:=TRUE;
                end;

        procedure def INIT(var PIL:PILE);
                begin
                PIL.TOP:=0;
                end;

?
```

Le programme suivant va utiliser des objets définis dans le programme ***PPILE*** *et contenus dans l'interface PPILEINTF.*

```
program CREEPILE(OUTPUT);
        var   I:INTEGER;
                PIL:PILE;
        begin
        INIT(PIL);
        I:=1;
        while I<=MAX do
                begin
                EMPILER(PIL,I*I);
                WRITE(SOMMET(PIL),' ');
                I:=I+1
                end
        end.
?.use
PPILEINTF
    05/19/83   14:52
    USE
    sur   PPILE

[lexp]:(ppileintf);
?p

(*USE(PPILEINTF)*)
program CREEPILE(OUTPUT);
```

```
      var   I:INTEGER;
            PIL:PILE;
      begin
      INIT(PIL);
      I:=1;
      while I<=MAX do
            begin
            #;#;#
            end
      end.
?.mcompile
CREEPILE.polish   FILE CREATED
Voulez-vous reduire l'interface globale? (oui/non) [line]:oui
Liste des objets a enlever de l'interface:
[lexp]:(i);
Quel nom voulez-vous donner a l'interface? [ident]:pilecarre
lline
PILECARRE.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN
lline
PILECARREC.polish   FILE CREATED
PILECARREC.pascal   FILE CREATED
PASCAL 6.02
?@interface p*
```

L'interface PILECARRE est une interface réduite sur le programme CREEPILE. L'entier I n'y figure pas. La réduction est présentée dans sa version usuelle, où il faut donner une liste des objets à enlever ou à laisser.

```
(*USE(PPILEINTF)*)
program PILECARRE(OUTPUT);
      var   PIL:PILE

.
?@compilable p*

program PILECARREC(OUTPUT,EMPILER,DEPILER,SOMMET,PILENONVIDE,INIT);
      const
            MAX=15;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:0..MAX
                  end;
      var   I:INTEGER;
            def PIL:PILE;
```

```
      procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

      procedure DEPILER(var PIL:PILE);external;

      function SOMMET(var PIL:PILE):INTEGER;external;

      function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

      procedure INIT(var PIL:PILE);external;

      begin
      INIT(PIL);
      I:=1;
      while I<=MAX do
            begin
            EMPILER(PIL,I*I);
            WRITE(SOMMET(PIL),' ');
            I:=I+1
            end
      end.
?.multics
[line]:pilecarrec
          1          4          9         16         25
         36         49         64         81        100
        121        144        169        196        225
?
```

La commande multics fait partie de l'environnement standard de Mentor et permet d'exécuter une commande Multics sans sortir de Mentor. Ici, elle permet d'exécuter le programme pilecarrec.

```
program HAUTEUR(OUTPUT);
      begin
      WRITELN('hauteur de PIL: ',PIL.TOP)
      end.
?.use
PPILEINTF
   05/19/83   14:52
   USE
   sur   PPILE

PILECARRE
   05/19/83   14:56
   USE(PPILEINTF)
   sur   CREEPILE
```

```
[lexp]:(ppileintf,pilecarre);
?p*

(*USE(PILECARRE,PPILEINTF)*)
program HAUTEUR(OUTPUT);
      begin
      WRITELN('hauteur de PIL: ',PIL.TOP)
      end.
?.mcompile
HAUTEUR.polish   FILE CREATED
Quel nom voulez-vous donner a l'interface? [ident]:haut
lline
HAUT.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN
lline
HAUTC.polish   FILE CREATED
HAUTC.pascal   FILE CREATED
PASCAL 6.02
?@compilable p*

program HAUTC(OUTPUT,PIL,EMPILER,DEPILER,SOMMET,PILENONVIDE,INIT);
      const
            MAX=15;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:0..MAX
                  end;
      var   PIL:PILE;

      procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

      procedure DEPILER(var PIL:PILE);external;

      function SOMMET(var PIL:PILE):INTEGER;external;

      function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

      procedure INIT(var PIL:PILE);external;

      begin
      WRITELN('hauteur de PIL: ',PIL.TOP)
      end.
?.load
File name:ppileintf
?p
```

```
(*USE*)
program PPILEINTF;
      const
            MAX=15;
      type PILE=...;

      procedure EMPILER(#;#);external;

      procedure DEPILER(#);external;

      function SOMMET(#):INTEGER;external;

      function PILENONVIDE(#):BOOLEAN;external;

      procedure INIT(#);external;

.
?.crintf
Voulez-vous reduire l'interface globale? (oui/non) [line]:oui
Liste des objets a enlever de l'interface:
[lexp]:;
Liste des objets a laisser dans l'interface:
[lexp]:(pile);
MAX    est automatiquement conserve.
Quel nom voulez-vous donner a l'interface? [ident]:ppileintf2
lline
PPILEINTF2.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN
?p*
```

L'interface PPILEINTF2 est obtenue par réduction cohérente de l'interface **PPILEINTF**.

```
(*USE*)
program PPILEINTF2;
      const
            MAX=15;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:0..MAX
                  end

.
?.load
File name:hauteur
?1p
(*USE(PILECARRE,PPILEINTF)*)
program HAUTEUR(OUTPUT)
```

```
?.use
PPILEINTF
    05/19/83   14:52
    USE
    sur   PPILE

PILECARRE
    05/19/83   14:56
    USE(PPILEINTF)
    sur   CREEPILE

HAUT
    05/19/83   14:59
    USE(PILECARRE,PPILEINTF)
    sur   HAUTEUR

PPILEINTF2
    05/19/83   15:03
    USE
    sur   PPILEINTF

[lexp]:(pilecarre,ppileintf2);
?p*

(*USE(PPILEINTF2,PILECARRE)*)
program HAUTEUR(OUTPUT);
      begin
      WRITELN('hauteur de PIL: ',PIL.TOP)
      end.
?.mcompile
Voulez-vous utiliser une interface existante? (oui/non) [line]:oui
lline
HAUTEUR.polish   FILE OVERWRITTEN
Donner le nom de l'interface qu'il faut utiliser pour
compiler le programme: [ident]:haut
Voulez-vous remplacer l'interface existante? (oui/non) [line]:non
CATALOGUE.polish   FILE OVERWRITTEN
lline
HAUTC.polish   FILE CREATED
HAUTC.pascal   FILE CREATED
PASCAL 6.02
?@compilable p*
```

La recompilation du programme HAUTEUR s'est faite en utilisant l'interface **HAUT**, *qui est encore une interface valide sur ce programme. L'utilisation de l'interface réduite* **PPILEINTF2** *a permis de réduire le contexte de compilation.*

```
program HAUTC(OUTPUT,PIL);
      const
            MAX=15;
      type PILE=
                  record
                        P:array[1..MAX]of INTEGER;
                        TOP:0..MAX
                  end;
      var   PIL:PILE;
      begin
      WRITELN('hauteur de PIL: ',PIL.TOP)
      end.
?.multics
[line]:hautc
hauteur de PIL:              15
?

?p*

program PILEQUATRE(OUTPUT);
      var   PILCA:PILE;
      begin
      INIT(PILCA);
      while PILENONVIDE(PIL)do
            begin
            EMPILER(PILCA,SOMMET(PIL)*SOMMET(PIL));
            DEPILER(PIL);
            WRITE(SOMMET(PILCA),' ')
            end
      end.
?.use
PPILEINTF2
   05/19/83   15:03
   USE
   sur   PPILEINTF

PPILEINTF
   05/19/83   14:52
   USE
   sur   PPILE

PILECARRE
   05/19/83   14:56
   USE(PPILEINTF)
   sur   CREEPILE
```

```
HAUT
   05/19/83   14:59
   USE(PILECARRE,PPILEINTF)
   sur   HAUTEUR

[lexp]:(ppileintf,pilecarre);
?p

(*USE(PILECARRE,PPILEINTF)*)
program PILEQUATRE(OUTPUT);
      var   PILCA:PILE;
      begin
      INIT(PILCA);
      while PILENONVIDE(PIL)do
            begin
             #;#
            end
      end.
?.mcompile
PILEQUATRE.polish   FILE CREATED
Voulez-vous reduire l'interface globale? (oui/non) [line]:non
Quel nom voulez-vous donner a l'interface? [ident]:quatre
lline
QUATRE.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN
lline
QUATREC.polish   FILE CREATED
QUATREC.pascal   FILE CREATED
PASCAL 6.02
?.multics
[line]:quatrec
      50625        38416        28561        20736        14641
      10000         6561         4096         2401         1296
        625          256           81           16            1
?.multics
[line]:hautc
hauteur de PIL:            0
```

L'interface virtuelle suivante définit une spécification du sous–programme **PILEPLEINE** *qui sera réalisée dans un programme ultérieur.*

```
?p*

program FAIREPILE;
```

```
      procedure PILEPLEINE(var PIL:PILE;IT:INTEGER);external;
      .
?.use
PPILEINTF2
   05/19/83   15:03
   USE
   sur   PPILEINTF

PPILEINTF
   05/19/83   14:52
   USE
   sur   PPILE

PILECARRE
   05/19/83   14:56
   USE(PPILEINTF)
   sur   CREEPILE

HAUT
   05/19/83   14:59
   USE(PILECARRE,PPILEINTF)
   sur   HAUTEUR

QUATRE
   05/19/83   15:08
   USE(PILECARRE,PPILEINTF)
   sur   PILEQUATRE

[lexp]:(ppileintf);
?p

(*USE(PPILEINTF)*)
program FAIREPILE;

      procedure PILEPLEINE(#;#);external;
      .
?.saveintf
FAIREPILE.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN

?p*

program REALISA(OUTPUT);
      var   PP:PILE;
```

```
    procedure PILEPLEINE;
        var  I:INTEGER;
        begin
        INIT(PIL);
        I:=IT;
        while I<IT+MAX do
            begin
            EMPILER(PIL,I);
            WRITE(SOMMET(PIL),' ');
            I:=I+1
            end;
        end;

    begin
    PILEPLEINE(PP,7)
    end.
?.use
PPILEINTF2
   05/19/83   15:03
   USE
   sur  PPILEINTF

PPILEINTF
   05/19/83   14:52
   USE
   sur  PPILE

PILECARRE
   05/19/83   14:56
   USE(PPILEINTF)
   sur  CREEPILE

HAUT
   05/19/83   14:59
   USE(PILECARRE,PPILEINTF)
   sur  HAUTEUR

QUATRE
   05/19/83   15:08
   USE(PILECARRE,PPILEINTF)
   sur  PILEQUATRE

FAIREPILE
   05/19/83   15:17
   USE(PPILEINTF)
   sur
```

```
[lexp]:(fairepile,ppileintf);
?1p
(*USE(FAIREPILE,PPILEINTF)*)
program REALISA(OUTPUT)
?.realize
REALISA.polish   FILE CREATED
Nom de l'interface virtuelle:
[ident]:fairepile
Nom du programme modulaire cree:
[ident]:rea
lline
REA.polish   FILE CREATED
CATALOGUE.polish   FILE OVERWRITTEN
?p*
```

Le programme REALISA réalise les spécifications contenues dans l'interface virtuelle **FAIREPILE**. *Le programme modulaire REA résultant de la commande realize va être compilé.*

```
(*USE(PPILEINTF)*)
program REA(OUTPUT);
      var   PP:PILE;

      procedure PILEPLEINE(var PIL:PILE;IT:INTEGER);forward;

      procedure PILEPLEINE;
            var   I:INTEGER;
            begin
            INIT(PIL);
            I:=IT;
            while I<IT+MAX do
                  begin
                  EMPILER(PIL,I);
                  WRITE(SOMMET(PIL),' ');
                  I:=I+1
                  end;
            end;

      begin
      PILEPLEINE(PP,7)
      end.
?.mcompile
lline
REA.polish   FILE OVERWRITTEN
Voulez-vous reduire l'interface globale? (oui/non) [line]:oui
Liste des objets a enlever de l'interface:
```

```
[lexp]:(pp);
Quel nom voulez-vous donner a l'interface? [ident]:rintf
lline
RINTF.polish  FILE CREATED
CATALOGUE.polish  FILE OVERWRITTEN
lline
RINTFC.polish  FILE CREATED
RINTFC.pascal  FILE CREATED
PASCAL 6.02
?.multics
[line]:rintfc
          7          8          9          10         11
         12         13         14         15         16
         17         18         19         20         21
?@compilable p*

program RINTFC(OUTPUT,EMPILER,DEPILER,SOMMET,PILENONVIDE,INIT);
    const
        MAX=15;
    type PILE=
            record
                P:array[1..MAX]of INTEGER;
                TOP:0..MAX
            end;
    var  PP:PILE;

    procedure EMPILER(var PIL:PILE;ITEM:INTEGER);external;

    procedure DEPILER(var PIL:PILE);external;

    function SOMMET(var PIL:PILE):INTEGER;external;

    function PILENONVIDE(var PIL:PILE):BOOLEAN;external;

    procedure INIT(var PIL:PILE);external;

    procedure def PILEPLEINE(var PIL:PILE;IT:INTEGER);forward;

    procedure def PILEPLEINE;
        var  I:INTEGER;
        begin
        INIT(PIL);
        I:=IT;
        while I<IT+MAX do
            begin
            EMPILER(PIL,I);
```

```
                WRITE(SOMMET(PIL),' ');
                I:=I+1
                end;
            end;

    begin
    PILEPLEINE(PP,7)
    end.
?.pcat
PPILEINTF2
   05/19/83   15:03
   USE
   sur   PPILEINTF

PPILEINTF
   05/19/83   14:52
   USE
   sur   PPILE

PILECARRE
   05/19/83   14:56
   USE(PPILEINTF)
   sur   CREEPILE

HAUT
   05/19/83   14:59
   USE(PILECARRE,PPILEINTF)
   sur   HAUTEUR

QUATRE
   05/19/83   15:08
   USE(PILECARRE,PPILEINTF)
   sur   PILEQUATRE

FAIREPILE
   05/19/83   15:17
   USE(PPILEINTF)
   sur   REALISA

RINTF
   05/19/83   15:20
   USE(PPILEINTF)
   sur   REA

?.end
EXIT LEV.1-PASCAL
```

2– Les programmes Mentol

VAL.mentol
CATALOG.mentol
CHECKINTF.mentol
FUSION.mentol
INTRED.mentol
INTREDRAPID.mentol
MCOMPILE.mentol
MERGEX.mentol
UTILITAIRES.mentol
MODULAIRE.messages

```
%
%
%                              PASCAL  MODULAIRE
%
%                                Valerie MIGOT
%
%
.def<.use<@use>,(.mentolin<@catalogue>;.use)>
.def<.triuse<@use,@tri>,(.mentolin<@catalogue>;.triuse<@use>)>
.def<.tri<@base,@nextkey,@place>,(.mentolin<@catalogue>;.tri)>
.def<.verif,(.mentolin<@mcompile>;.verif)>
.def<.mcompile<@monprog,@nprg,@yy>,(.mentolin<@mcompile>;.mcompile)>
.def<.crintf<@nom,@nomp>,(.mentolin<@mcompile>;.crintf)>
.def<.progintf<@itf,.byref<@pr>,@iif>,(.mentolin<@checkintf>;.progintf<@itf,@pr
\c>)>
.def<.checkintf<@contexte>,(.mentolin<@checkintf>;.checkintf)>
.def<.checkuseintf<@use2,@listutil>,(.mentolin<@checkintf>;.checkuseintf)>
.def<.lintf<@prog,.byref<@lintf>>,(.mentolin<@utilitaires>;.lintf)>
.def<.ecrit<@pmx>,(.mentolin<@utilitaires>;.ecrit)>
.def<.aide,(.mentolin<@utilitaires>;.aide)>
.def<.detail<@n,@command>,(.mentolin<@utilitaires>;.detail)>
.def<.liste,(.mentolin<@utilitaires>;.liste)>
.def<.concate<@ligne,@iidd,.byref<@res>>,(.mentolin<@utilitaires>;.concate<@lig
\cne,@iidd,@res>)>
.def<.left<@x,.byref<@xx>>,(.mentolin<@utilitaires>;.left<@x,@xx>)>
.def<.date<@xln,.byref<@xiddd>>,(.mentolin<@utilitaires>;.date<@xln,@xiddd>)>
.def<.intfbind<@nomc>,(.mentolin<@utilitaires>;.intfbind)>
.def<.bind<@x,@bibi>,(.mentolin<@utilitaires>;.bind<@x>)>
.def<.chcat,(.mentolin<@utilitaires>;.chcat)>
.def<.saveintf<@qwx,@sov>,(.mentolin<@fusion>;.saveintf<@qwx>)>
.def<.realize<@x,@y,@xnom,@ynom>,(.mentolin<@fusion>;.realize)>
.def<.fusionintf<@nomintf,@listintf,@uuse,@interf,@mdle,@svprg>,(.mentolin<@fus
\cion>;.fusionintf)>
.def<.reduc,(.mentolin<@intred>;.reduc)>
.def<.metdef<@lextw>,(.mentolin<@mergex>;.metdef)>
.def<.merge<@use,@interf,@mdle>,(.mentolin<@mergex>;.merge)>
.def<.fusionsimple<@pp,@nom>,(.mentolin<@mergex>;.fusionsimple)>
.def<.espion<"compiled>,"espion>
.def<.polish,(@ookk c @non)>
.def<.listing,(@ookk c @oui)>
.def<.wizard,(@intred c @intredrapid)>
.def<.bavard,(@intred c @intredlent)>

.rec
editeur
:&
.def<.fin,
      (.xin;
       .espion;
       @mentolin 1 i @stdenvl;
       @ookk = @non;
       .prmess<"modulaire,1>;
       @kenom c &;
       @catalog = @kenom;
```

```
      .pcat )>
 %
 %permet de sortir du prelude et de charger le catalogue
 %pour la session de travail

ss2;.ldef
:&
.def<.pcat,
      (@cat 1;?@cat u
              ,(pd;
                .load<@catalog>;?(@cat : @k;
                                   pu)
                                 ,(.prmess<"modulaire,2>;
                                   pu;
                                   $3));
       .apl2<@cat,(1 p;
                   .pnnl<@blancligne>;
                   b0 p;
                   .pnnl<@blancligne>;
                   b1 p;
                   .pnnl<@blancligne>;
                   .pnnl<@sur>;
                   .is<2 2,@meta>;?2 1 b1 p
                                   ,2 2p;
                   @blancligne p)>)>
 %
 %ecrit le catalogue

ss2;.ldef
.end
@soulq : &
lline]
$
_*)
.charid<@soulq,+,0>
.charid<@soulq,+,1>
@catalogue : &
ident]
catalog
@checkintf : &
ident]
checkintf
@fusion : &
ident]
fusion
@intred : &
ident]
intredrapid
@intredrapid : &
ident]
intredrapid
@intredlent : &
ident]
intred
@mcompile : &
ident]
```

```
mcompile
@mergex : &
ident]
mergex
@utilitaires : &
ident]
utilitaires
@kenom = @call
@kenom 1
@blancligne : &
lline]
   *)
@ext1 : &
body]
external;
@oui : &
line]
oui
@non : &
line]
non
@repws : &
ident]
oui;
@repws b0 c &
oui *)
@reponse : @repws b0 1
@sur : &
lline]
sur  *)
@cat0 : &
lexp]
($ident1 (of_,$ident2));
@estgarde : &
lline]
   est automatiquement conserve.  *)
.xin
```

CATALOG.mentol

```
 %
 %
 %                          PASCAL  MODULAIRE
 %
 %                            Valerie MIGOT
 %
 %
.rec
editeur
:&
.def<.use<@use>,
      (1 b0 c @listuse;
```

```
       .pcat;
       1 b0 2 c &;
       @use:@k 1 b0 2;
       @use 2 ;?(@use u;
                 .triuse<@use>)
                ,(@cat f @use 1;?@cat u*,(d @use 1;?)))>
 %
 %permet a l'utilisateur d'ecrire la liste des interfaces
 %utilisees par le programme courant en commentaire prefixe
 %du premier fils de @k
 %on obtient ainsi:
 %(*USE(A,B,C)*)
 %program D;
 %la liste A,B,C est ordonnee par .triuse
 %de telle facon que A peut contenir B ou C dans sa liste use
 %A,B et C sont des noms d'interfaces

ss2;.lredef
:&
.def<.triuse<@use,@tri>,
      (@tri=@lexp;
       @cat 1;?,(d @use; $2);
       @tri 1;
       (@use f @cat 1;?(@tri j @cat 1;
                        @use u;
                        @tri 1);
        @cat r)*;?;
       @cat u;
       @tri u;
       d @tri s*;
       @use c @tri)>
 %
 %@use est la liste des interfaces qu'utilise le programme
 %courant
 %@tri contient la liste triee obtenue a partir du
 %catalogue en ne gardant que les noms apparaissant
 %dans @use et en les mettant dans l'ordre inverse de celui
 %dans lequel ils etaient dans le catalogue
 %remarque:
 %le catalogue lu de gauche a droite donne un ordre possible
 %entre les interfaces et les programmes qu'il contient car
 %les interfaces sont inserees a droite dans le catalogue
 %lors de leur creation, sachant qu'un programme ne peut etre
 %compile que si toutes les interfaces qu'il utilise existent
 %donc sont situees a sa gauche dans le catalogue

ss2;.lredef
:&
.def<.lti<@xintf,@yintf,@copycat>,
      (@copycat = @cat;
       @copycat f @xintf;?(@copycat u bl;
                           @copycat f @yintf;?(@copycat u* u0 u*;
                                               $-3);
                           @copycat u* u0 u*))>
 %
 %@xintf et @yintf designent deux interfaces du catalogue
```

```
 %lti est vraie si @xintf n'utilise pas @yintf ie @xintf
 %est a gauche de @yintf

ss2;.ldef
:&
.def<.tri<@base,@nextkey,@place>,
    (kpd;@base:@cat s;
        ?,$;
        (k:@base r;?,$;
         l;@nextkey:k;k:@base;l;
         .lti<@nextkey,k>;
            ?(@place:@base;
                (@place l;
                    ?,(@place jd@base r;$2);
                 k:@place;l;
                 .lti<@nextkey,k>/;
                 @place id@base r;$)*),
             @base r)*;
        ?kpu,(kpu;$-))>
 %
 %tri le catalogue en respectant l'ordre suivant:
 %une interface ne peut utiliser que des interfaces
 %dont le nom est situe a sa gauche dans le catalogue

ss2;.lredef
:&
.def<.range<@x,@xx>,
      (@cat l l;?(@cat u2;
                  .ajoute)
                 ,.crecat;
       .left<@x,@xx>;
       @cat s* l c @xx;
       @cat s* b0 c @x b0 3;
       @cat s* bl c @x l b0)>
 %
 %@x designe une interface
 %si le catalogue contenu dans @cat est vide
 %alors .crecat
 %sinon .ajoute

ss2;.ldef
:&
.def<.crecat,
      (@cat = @cat0)>
 %

ss2;.ldef
:&
.def<.ajoute,
      (@cat s* i @cat0 1)>
 %

ss2;.ldef
:&
.def<.cycle<@x,@usse>,
      (@usse 1;?(@usse u;
```

```
                .apl2<@usse,.cycle1<@x,@k>>)>
 %
 %@usse contient la liste use de l'interface courante
 %dont le nom est dans @x
 %.cycle execute .cycle1 pour chaque nom d'interface dans @usse

ss2;.ldef
:&
.def<.cycle1<@x,@y>,
      (@cat f @x;
       @cat u;
       @cat l;?
              ,(@cat u*;
                $-2);
       (.eqval<@cat l,@y>;?$,@cat l)*;?
                                    ,(@cat u*;$-2);
       @cat u*)>
 %
 %.cycle1 verifie que le catalogue est bien ordonne pour @x
 %c'est-a-dire que pour une interface dont le nom (@x) est
 %dans le catalogue, les noms d'interfaces de sa liste use
 %se trouvent a sa gauche dans le catalogue
 %(ils ont ete compiles avant le programme qui les utilise)

ss2;.ldef
:&
.def<.catgus<@calle>,
     (@gus = @lexp;
      @calle = @call;
      @gus 1 c @calle;
      .apl2<@cat,(@gus s* i bl;@gus s* 1 c 1;
                  .is<@ident,2 2>;
                     ?(@gus s* i 2 bl;
                       @gus s* 1 c 2 2;
                           ?,(.prmess<"modulaire,49,2 2>;$-4)))>;
      d @gus 1)>

ss2;.ldef
:&
.def<.avechild<"compiled>,"ordon_ic>

ss2;.ldef
:&
.def<.querecomp<@xxx,@gus>,
      (.catgus;
       .avechild;
       @arecompiler = @lexp;
       @arecomp 1 1;?(@arecomp u*;
                      .apl2<@arecomp,@arecompiler s* ii 2>;
                      (d @arecompiler 1)2)
                    ,(@arecompiler : @arecomp;
                      d @arecompiler 1);
       (@arecompiler f @xxx;?d @arecompiler,$+1)*;
       @arecompiler 1;?(@arecompiler u;
                        .prmess<"modulaire,24>;
                        @arecompiler p;
```

```
                        .prmess<"modulaire,15>;
                        .invalidation)
                      ,.prmess<"modulaire,25>)>
 %
 %utilise le programme pascal ORDON_IC pour donner une liste
 %ordonnee des interfaces invalideès, cette liste se trouve
 %dans la variable @arecompiler

ss2;.lredef
:&
.def<.invalidation,
      (.apl2<@arecompiler,(@cat f @k;
                           @cat u;
                           @cat b0 c @invalidation;
                           (@cat r;?,(@cat u;@cat r);
                            @cat ff @k;
                                ?(@cat u;
                                  @cat b0 c @invalidation))*;?;
                           @cat u*)>)>
 %
 %dans le catalogue, remplace la date des composants invalides
 %par ´invalide´

ss2;.lredef
.end
@listuse = @call
@listuse 1 c &
use
@invalidation :&
line]
invalide
$
.xin
```

CHECKINTF.mentol

```
 %
 %
 %                        PASCAL  MODULAIRE
 %
 %                          Valerie MIGOT
 %
 %
.rec
editeur
:&
.def<.progintf<@itf,.byref<@pr>>,
      (@cat f @itf;
       @cat u;
       @cat 2 2;?
                ,(@cat u*;
                  $-2);
```

```
       @pr = @cat;
       @cat u*)>
 %
 %@itf designe le nom d'une interface pour laquelle
 %on veut connaitre le nom du programme correspondant

ss2;.lredef
:&
.def<.defctxl,
      (@contextel = @lident;
       .con;?(.apl<@contextel s* i @k 1>);
       u*;
       .typ;?(.apl<(@contextel s* i @k 1;
                    .is<@lident,@k 2>;?@contextel s* ii @k 2)>);
       d @contextel 1;
       u*)>
 %
 %@k designe une interface
 %@contextel designe les noms des constantes
 %et des types qui y sont definis

ss2;.ldef
:&
.def<.defctx<@contextel,@use,@interfl,@progsv>,
      (.defctxl;
       @contexte = @contexte0;
       @contexte s* ii @contextel;?;
       @contexte01 = @contexte;
       @use = @k 1 b0 2;?(@use 1;?
                          (@progsv:@k;
                           (@interfl : @use;
                            .load<@interfl>;?
                                     ,(.prmess<"modulaire,3,@interfl>;
                                       pu;
                                       $-4);
                            .defctxl;
                            @contexte s* ii @contextel;?;
                            @use r)*;
                           ?;
                           @k:@progsv;
                           @use u))
                        ,.prmess<"modulaire,4>)>
 %
 %@k designe une interface
 %@contexte contiendra la liste de constantes et de types
 %appartenant au contexte de compilation de cette interface

ss2;.ldef
:&
.def<.examen<@arbre>,
    (.is<@ident,@arbre>/(@contexte f @arbre;
                         ?@contexte u
                         ,.prmess<"modulaire,26,@arbre>);
     .is<@lident,@arbre>/;
     .is<@lfield,@arbre>/
                .apl2<@arbre,(.is<@decl,@k>;
```

```
                                        ?.examen<2>
                                        ,(.examen<1 2>;
                                             ?,.examen<1>;
                                          .apl2<2,(.examen<1>;
                                                   .apl2<2,.examen<2>>)>)>)>;
      .is<@array,@arbre>/(.apl2<@arbre 1,.examen<@k>>;
                          .examen<@arbre 2>);
      .is<@range,@arbre>/(.examen<@arbre 1>;
                          .examen<@arbre 2>);
      @arbre 1/(.examen<@arbre>;
                @arbre u))>
 %
 %@arbre contient la description d'un type
 %on verifie si les objets utilises dans la description
 %de ce type appartiennent au contexte de compilation de
 %l'interface contenant la declaration de ce type

ss2;.ldef
:&
.def<.checkintf<@contexte>,
      (u*;
       2 1;?(u*;
             .defctx;
             .con;?(.apl<(2;
                          .is<@ident>;
                              ?(@contexte f @k;
                                   ?@contexte u
                                   ,.prmess<"modulaire,26,@k>)>)>);
             u*;
             .typ;?(.apl<.examen<@k 2>>);
             u*;
             .var;?(.apl<.examen<@k 2>>);
              u*;
              2;
              .foreach<@block,(1 2;
                               1;?(u;
                                   .apl<.examen<@k 2>>))>;
             u*))>
 %
 %verifie la coherence de l'interface designee par @k
 %teste si les constantes et les types utilises appartiennent
 %au contexte de compilation de l'interface (ou du programme)

ss2;.lredef
:&
.def<.defctx2<@contextel,@use,@interfl,@progsv>,
      (@use = @k 1 b0 2;
            ?(@use 1;?(@progsv:@k;
                       (@interfl : @use;
                        .load<@interfl>;?
                                ,(.prmess<"modulaire,3,@interfl>;
                                  pu;
                                  $-4);
                        .defctxl;
                        .utility<@listutil,@contextel>;
                        @use r)*;
```

```
                        ?;
                        @k:@progsv;
                        @use u))
            ,.prmess<"modulaire,4>)>
 %
 %@k designe une interface
 %@contexte contiendra la liste de constantes et de types
 %appartenant au contexte de compilation de cette interface

ss2;.ldef
:&
.def<.utility<@lis,@cont>,
      (@cont 1;?(@contexte01 f @cont;
                      ?(@contexte01 u;
                        @cont r)
                      ,((@lis f @cont;?(@use2 s* i @use;
                                        @cont u;
                                        @lis u;
                                        $+4)
                                       ,@cont r)*;?;
                        @cont u)))>
 %
 %

ss2;.ldef
:&
.def<.exa2<@arbre>,
     (.is<@ident,@arbre>/
               (@contexte f @arbre;?(@contexte u;
                                     @listutil s* i @arbre)
                                    ,.prmess<"modulaire,26,@arbre>);
      .is<@lident,@arbre>/;
      .is<@lfield,@arbre>/
               .ap12<@arbre,(.is<@decl,@k>;?.exa2<2>
                                    ,(.exa2<1 2>;?
                                                ,.exa2<1>;
                                      .ap12<2,(.exa2<1>;
                                                .ap12<2,.exa2<2>>)>)>)>;
      .is<@array,@arbre>/(.ap12<@arbre 1,.exa2<@k>>;
                          .exa2<@arbre 2>);
      .is<@range,@arbre>/(.exa2<@arbre 1>;
                          .exa2<@arbre 2>);
      @arbre 1/(.exa2<@arbre>;
                @arbre u))>
 %
 %@arbre contient la description d'un type
 %on verifie si les objets utilises dans la description
 %de ce type appartiennent au contexte de compilation
 %de l'interface contenant la declaration de ce type

ss2;.ldef
:&
.def<.checkintf2<@contexte,@contexte01>,
      (u*;
       2 1;?(u*;
             .defctx;
```

```
            .con;?(.apl<(2;
                        .is<@ident>;
                            ?(@contexte f @k;
                                  ?(@contexte u;
                                    @listutil s* i @k)
                                  ,.prmess<"modulaire,26,@k>)))>);
            u*;
            .typ;?(.apl<.exa2<@k 2>>);
            u*;
            .var;?(.apl<.exa2<@k 2>>);
             u*;
             2;
             .foreach<@block,(1 2;
                              1;?(u;
                                  .apl<.exa2<@k 2>>)))>;
            u*;
            .defctx2))>
 %
 %verifie la coherence de l'interface designee par @k
 %teste si les constantes et les types utilises appartiennent
 %au contexte de compilation de l'interface (ou du programme)

ss2;.ldef
:&
.def<.checkuseintf<@use2,@listutil>,
      (@use2 = @lexp;
       @listutil = @lident;
       .checkintf2;
       d @use2 1;
       .triuse<@use2>;
       @k 1 b0 2 c @use2)>
 %
 %@k designe une interface
 %verifie la coherence de l'interface et
 %reduit la liste USE de l'interface si possible

ss2;.lredef
.end
@contexte0 : &
lident]
(integer,real,char,boolean,true,false,text,alfa,alfaleng);
$
.xin
```

FUSION.mentol

```
 %
 %
 %                          PASCAL  MODULAIRE
 %
 %                            Valerie MIGOT
```

```
%
%
.rec
editeur
:&
.def<.saveintf<@qwx,@sov>,
      (@qwx 1;
            ?(@qwx u;
              pd;
              @k : @qwx;
              .verif;
              .eqval<@ookk,@non>;?.sauve,.write;
              pu;
              .range<@qwx>)
            ,(.verif;
              .eqval<@ookk,@non>;?.sauve,.write;
              .range<@k>);
       @sov = @k;
       @k : @cat;
       .store<@catalog>;
       @k:@sov)>
%
%@qwx designe une interface qui vient d'etre ecrite
%.saveintf sauvegarde cette interface sous forme
%polish et/ou pascal,
%met a jour le catalogue et le sauve.
%si @qwx est omis, on prend par defaut @k

ss2;.lredef
:&
.def<.realize<@x,@y,@xnom,@ynom,@mark>,
      (.verif;
       .eqval<@ookk,@non>;?.sauve,.write;
       @y=@k;
       .prmess<"modulaire,29>;
       .leftms;
       (@kenom c &;
        @cat f @kenom ; ?(@cat u*;$2);
        .prmess<"modulaire,10,@kenom>)*;
       @xnom = @kenom;
       @ynom = @k;
       .prmess<"modulaire,30>;
       @kenom c &;
       @k c @kenom;
       u*;
       @realisation = @k;
       .load<@xnom>;
       .foreach<@ext1,(@mark = 12;
                       @mark 1;
                       @y f @mark;
                          ?(@y u*;c @forward)
                          ,(@k : @y;.prmess<"modulaire,31>;$-3))>;
       @x = @k;
       @k = @realisation;
       .fusionuse<@xnom,@ynom,@x 1 b0 2,@k 1 b0 2>;
       @k 2 1;?(u*;
```

```
                .mergeconst<@x>;
                .mergetype<@x>;
                .mergevarsansref<@x>;
                .mergefprocsanref<@x>)
              ,@k 2 c @x 2;
       u*;
       @cat f @xnom;
       @cat u;
       @cat 2 2 c @ynom;
       @cat 2 b0 c @y b0 3;?;
       @cat 2 b1 c @y 1 b0;
       @cat u;
       .eqval<@ookk,@non>;?.sauve,.write;
       @realisation : @k;
       @k : @cat;
       .store<@catalog>;
       @k : @realisation)>
%
%@y designe le programme qui realise l'interface contenue dans @x
%on fusionne les listes use du programme et de l'interface,
%dans l'interface on remplace les 'external' par 'forward'
%on insere dans le programme:
%-les constantes declarees dans l'interface (qui ne doivent pas
%etre repetees dans le programme source)
%-les types declares dans l'interface
%-les variables
%-les entetes de procedures et fonctions dont le corps est
%'forward' et dont la declaration complete est dans le
%programme source
%le programme ainsi obtenu sera alors mcompilable

ss2;.lredef
:&
.def<.fusionuse<@p0,@p1,@lp0,@lp1>,
      (@lp1 f @p0;?d @lp1;
       @lp0 f @p1;?d @lp0;
       @lp0 1;?@lp0 u
             ,$1;
       @lp1 1;?@lp1 u
             ,(@lp1 = @lp0;
               $2);
       @lp1 1 jj @lp0;
       .triuse<@lp1>)>
%
%fusionne deux listes use
%@p0 contient le nom du programme dont la liste use est dans @lp0
%de meme pour @p1 et @lp1
%@use contient le resultat de la fusion des deux listes @lp0 et @lp1
%obtenu en inserant @lp0 en tete de @lp1 et en triant la liste
%resultant a l'aide de .triuse

ss2;.ldef
:&
.def<.fusionintf<@nomintf,@listintf,@uuse,@interf,@mdle,@svprg>,
      (@listintf = @call;
       @nomintf : @listintf 1;
```

```
        @listintf 2;
        .prmess<"modulaire,27>;
        @listintf c &;
        .triuse<@listintf>;
        (.prmess<"modulaire,28>;
         @nomintf c &;
         @cat f @nomintf ; ?(.prmess<"modulaire,11>;
                             @cat u*)
                           ,$1)*;
        @uuse = @listintf;
        @uuse 1;
        @interf : @uuse;
        .load<@interf>;
        @listintf s* ii @k 1 b0 2;?;
        @uuse r;
        (@svprg = @k;
         @interf : @uuse;
         .load<@interf>;
         @listintf s* ii @k 1 b0 2;?;
         @mdle = @k;
         @k = @svprg;
         .merge0sansref<@mdle>;
         @uuse r)*;?;
        @uuse u;
        u*;
        .prmess<"modulaire,6>;
        (@reponse c &;
         .eqval<@reponse,@oui> / (.reduc;$2);
         .eqval<@reponse,@non> / $1;
         .prmess<"modulaire,5>)*;
        .leftms;
        c @nomintf;
        u*;
        @k 1 b0 2 c @listintf;
        .checkuseintf;
        .eqval<@ookk,@non>;?.sauve,.write;
        .range<@k>;
        @cat f @nomintf;
        @cat r;
        @cat 1 b1 c @uuse;
        @cat u*;
        pd;
        @k : @cat;
        .store<@catalog>;
        pu)>
 %
 %

ss2;.lredef
.end
.mentolin<@catalogue>
.mentolin<@mergex>
$
.xin
```

```
%
%
%                          PASCAL . MODULAIRE
%
%                            Valerie MIGOT
%
%
.rec
editeur
:&
.def<.reduc,
      (2 f @block;?
            (.prmess<"modulaire,32>;
             (@reponse c&;
              .eqval<@reponse,@oui> / $1;
              .eqval<@reponse,@non> /
                   (.prmess<"modulaire,42>;
                    (@reponse c&;
                     .eqval<@reponse,@oui> / (.reductoutfproc;$2);
                     .eqval<@reponse,@non> / (.reducfproc;$2);
                     .prmess<"modulaire,5>)*;
                    $2);
              .prmess<"modulaire,5>)*);
       u*;
       .var;?(.prmess<"modulaire,33>;
              (@reponse c&;
               .eqval<@reponse,@oui> / $1;
               .eqval<@reponse,@non> /
                    (.prmess<"modulaire,43>;
                     (@reponse c &;
                      .eqval<@reponse,@oui> / (.reductoutvar;$2);
                      .eqval<@reponse,@non> / (.reducvar;$2);
                      .prmess<"modulaire,5>)*;
                     $2);
              .prmess<"modulaire,5>)*);
       u*;
       .typ;?(.prmess<"modulaire,34>;
              (@reponse c&;
               .eqval<@reponse,@oui> / $1;
               .eqval<@reponse,@non> /
                    (.prmess<"modulaire,44>;
                     (@reponse c&;
                      .eqval<@reponse,@oui> / (.reductoutl;$2);
                      .eqval<@reponse,@non> / (.reducl;$2);
                      .pnnl<@ouinon>)*;
                     $2);
              .prmess<"modulaire,5>)*);
       u*;
       .con;?(.prmess<"modulaire,35>;
              (@reponse c&;
               .eqval<@reponse,@oui> / $1;
```

```
                .eqval<@reponse,@non> /
                     (.prmess<"modulaire,45>;
                      (@reponse c &;
                       .eqval<@reponse,@oui> / (.reductoutl;$2);
                       .eqval<@reponse,@non> / (.reducl;$2);
                       .prmess<"modulaire,5>)*;
                      $2);
                .prmess<"modulaire,5>)*);
       u*)>
 %
 %@k designe l'interface d'un programme
 %.reduc construit interactivement une interface reduite
 %en questionnant l'utilisateur sur ce qu'il veut laisser
 %parmi les fonctions,procedures,variables,types et
 %constantes

ss2;.lredef
:&
.def<.reductoutfproc,
      ((d r)*;?;
       d;
       u*)>
 %
 %enleve tous les sous-programmes du programme quand on s'est
 %place sur la premiere declaration de fonction ou procedure

ss2;.ldef
:&
.def<.reducfproc<@ptr>,
      ((.is<@def,@k 1 1>;?(r;?,$3)
        ,(.prmess<"modulaire,36,@k 1 1>;
          (@reponse c&;
           .eqval<@reponse,@oui> / (r;?$3,$5);
           .eqval<@reponse,@non> / (@ptr:@k;
                                    r;?(d @ptr;$4)
                                      ,(u;
                                        d @ptr;
                                        $6));
           .prmess<"modulaire,5>)*))*)>
 %
 %enleve les procedures et les fonctions

ss2;.ldef
:&
.def<.reductoutvar,
      (d;
       u*)>
 %
 %lorsqu'on se trouve sur la zone des variables
 %supprime toutes ces declarations

ss2;.ldef
:&
.def<.reducvar<@ptt>,
      (.apl<.traiter>;
       d @ptt;
```

```
      .eqval<@k,@null>;?d)>
%
%enleve les variables qu'on ne veut pas dans l'interface

ss2;.ldef
:&
.def<.traiter<@ptr>,
      (l;
       .apl<(.is<@def,@k>;?,
             (.prmess<"modulaire,36,@k>;
              (@reponse c&;
               .eqval<@reponse,@oui> / $1;
               .eqval<@reponse,@non> / (d @ptr;
                                        @ptr:@k;
                                        $2);
               .prmess<"modulaire,5>)*))>;
       d @ptr;
       .eqval<@k,@null>;?(u;
                          d @ptt;
                          @ptt:@k))>
%

ss2;.ldef
:&
.def<.reductoutl<@ptr,@trc>,
      (s*;
       (@trc : l;
        .pnnl<r ff @trc>;
                ?(@estgarde p;
                  l;?
                    ,$3)
                ,(.pnnl<u r ff @trc>;?(@estgarde p;
                                       l;?
                                         ,$4)
                                      ,(@ptr: @k;
                                        l;?d @ptr
                                          ,(u;
                                            d @ptr;
                                            .eqval<@k,@null>;?d;
                                            $5))))*)>
%
%

ss2;.ldef
:&
.def<.reducl<@ptr,@trc>,
      (s*;
       (@trc : l;
        .pnnl<r ff @trc>;?(@estgarde p;
                           l;?
                             ,$3)
                          ,(.pnnl<u r ff @trc>;
                              ?(@estgarde p;
                                l;?,$4)
                              ,(.prmess<"modulaire,36,@k 1>;
                                (@reponse c&;
```

```
                                .eqval<@reponse,@oui> / (1;?$2,$6);
                                .eqval<@reponse,@non> /
                                          (@ptr:@k;
                                           1;?(d @ptr;$3)
                                             ,(u;
                                               d @ptr;
                                               .eqval<@k,@null>;?d;
                                           $7));
                                .prmess<"modulaire,5>)*)))*)>
 %
 %permet d'enlever parmi les constantes ou les types
 %les objets que l'on ne veut pas voir apparaitre dans
 %l'interface
 %pour les types ou les constantes,on examine chaque declaration
 %en commencant par la derniere,en remontant,et en verifiant
 %si l'objet declare est utilise ulterieurement,
 %dans ce cas l'utilisateur ne peut pas enlever cet objet

ss2;.ldef
.end
$
.xin
```

INTREDRAPID.mentol

```
 %
 %
 %                        PASCAL  MODULAIRE
 %
 %                          Valerie MIGOT
 %
 %
.rec
editeur
:&
.def<.reduc<@listenlever,@listlaisser,@listobjet,@imcc>,
      (@listenlever = @call;
       @listenlever 2;
       .prmess<"modulaire,38>;
       @listenlever c &;
       @listenlever 1;?(.listobj;
                        @imcc = @lexp;
                        (@listobjet f @listenlever;
                              ?@listobjet u
                              ,@imcc s* i @listenlever;
                         @listenlever r)*;?;
                        @listenlever u;
                        d @imcc 1;
                        @imcc 1;
                         ?(@imcc u;
                          .complement<@listenlever,@imcc,@listenlever>);
                        @listenlever 1;?.reducenlever)
```

```
                        ,(.prmess<"modulaire,39>;
                          @listlaisser = @call;
                          @listlaisser 2;
                          @listlaisser c &;
                          @listlaisser 1;
                           ?(.listobj;
                             @listlaisser u;
                             .complement<@listobjet,@listlaisser,@listenlever>;
                             @listenlever 1;?.reducenlever)
                            ,d 2))>
 %
 %

ss2;.1redef
:&
.def<.reducenlever<@obj>,
      ((@k f @listenlever;?((.is<@def,@k u> / $2;
                             .is<@proc,@k u> /
                                 (u 2;d;$3);
                             .is<@func,@k u> /
                                 (u 2;d;$3);
                             .is<@ldefid,@k u> /
                                 (d;1;?,(u;d);$3);
                             .is<@eqlt,@k u> /
                                 (@obj = @k;
                                  .pnnl<r ff @obj>;
                                      ?@estgarde p
                                      ,(.pnnl<u r ff @obj>;
                                            ?@estgarde p
                                            ,(u;d));
                                  $3);
                             .is<@eqlc,@k u> /
                                 (@obj = @k;
                                  .pnnl<r ff @obj>;
                                      ?@estgarde p
                                      ,(.pnnl<u r ff @obj>;
                                            ?@estgarde p
                                            ,(u;d));
                                  $3);
                            @k u;
                            @k r;?,(@k u;@k r;?,(@k u;@k r));
                            @k ff @listenlever)*;?);
        @k u*;
        @listenlever r)*;?;
        2;
        (1 1;?(u2;$+2),d 1)*;?;
        (2 1;?(u2;$+2),d 2)*;?;
        (3 1;?(u2;$+2),d 3)*;?;
        u)>
 %
 %@listenlever contient les objets que l'on veut supprimer
 %de l'interface, les types et les constantes doivent etre
 %a la fin de cette liste, sinon ils peuvent etre conserves

ss2;.1def
:&
```

```
.def<.complement<@biglist,@sublist,.byref<@result>>,
      (@result = @biglist;
       .apl2<@sublist,(@result f @k;?d @result)>)>
 %
 %@sublist est une sous-liste de @biglist
 %@result U @sublist = @biglist
 %si @sublist contient par erreur des objets n'appartenant
 %pas a @biglist, ces objets ne seront pas dans @result

ss2;.ldef
:&
.def<.listobj,
      (2;
       @listobjet = @ldefid;
       .foreach<@block,@listobjet s* i @k 1 1>;
       u*;
       .var;?.apl<@listobjet s* ii @k 1>;
       @listobjet 1;
       (@listobjet r;
        .is<@def,@listobjet>;?@listobjet c @listobjet 1)*;?;
       @listobjet u;
       u*;
       .typ;?.apl<@listobjet s* i @k 1>;
       u*;
       .con;?.apl<@listobjet s* i @k 1>;
       u*;
       d @listobjet 1)>
 %
 %

ss2;.ldef
.end
$
.xin
```

MCOMPILE.mentol

```
 %
 %
 %                          PASCAL  MODULAIRE
 %
 %                            Valerie MIGOT
 %
 %
.rec
editeur
:&
.def<.mcompile<@monprog,@nprg,@yy>,
      (u*;
       .verif;
       .leftms;
       @monprog = @k;
```

```
       @nprg : @k;
       u*;
       @cat f @monprog;?(@cat u*;
                         .recompile<@monprog,@nprg>)
                        ,.mecompile<@monprog,@nprg>;
       .eqval<@ookk,@non>;?(.concat<@delete 2,@v,@yy>;
                            .concat<@yy,@pppp,@yy>;
                            .os<@yy>);
       @compilable:@k;
       @k:@source)>
 %
 %@k designe le programme qu'on veut compiler
 %@monprog designe le nom du programme
 %si le nom du programme n'est pas dans le catalogue , il n'a
 %pas encore ete compile, on lance alors la commande
 %.mecompile
 %sinon on recompile le programme par la commande
 %.recompile

ss2;.1redef
:&
.def<.mecompile<@monprog,@nprg,@nyface>,
      (.mcompilel<@nprg>;
       .load<@monprog>;
       .miseajour<@k>;
       .metdef;
       .merge;
       .compile)>
 %
 %enchaine toutes les procedures pour
 %compiler un programme pour la premiere fois

ss2;.1def
:&
.def<.mcompilel<@x>,
      (@k u*;
       @source=@k;
       .eqval<@ookk,@non>;?.sauve,.write;
       @k 2 1;?(u*;
                .iface;
                .prmess<"modulaire,6>;
                (@reponse c&;
                 .eqval<@reponse,@oui> / (.reduc;$2);
                 .eqval<@reponse,@non> / $1;
                 .prmess<"modulaire,5>)*)
               ,d 3;
       .leftms;
       (.prmess<"modulaire,21>;
        @kenom c &;
        @nyface = @kenom;
        .eqval<@nyface,@monprog> / @egnomprog p;
        @cat f @nyface;?(.prmess<"modulaire,11>;
                         @cat u*)
                        ,$1)*;
       @k c @nyface;
       u*;
```

```
       .range<@k>;
       @cat f @nyface;
       @cat u;
       @cat 2 2 c @monprog;
       @cat u*;
       u*;
       @interface:@k;
       .ecrit;
       @k : @cat;
       .store<@catalog>)>
%
%ecrit le programme source sous forme polish et/ou pascal,
%le range dans le catalogue,
%cree l'interface de ce programme,
%sauvegarde le catalogue dans NOM.polish
%ou NOM est l'identificateur designe par la variable @catalog

ss2;.ldef
:&
.def<.recompile<@monprog,@nprg,@nyface,@np,@qq>,
      (.prmess<"modulaire,9>;
       (@reponse c &;
        .eqval<@reponse,@oui> /
                  (.mcompile2<@nprg>;
                   .load<@monprog>;
                   .metdef;
                   .merge;
                   .concat<@delete 2,@np,@qq>;
                   .concat<@qq,@delete 1,@qq>;
                   .os<@qq>;
                   .compile;
                   $2);
         .eqval<@reponse,@non> /
                   (.mecompile<@monprog,@nprg>;
                    $2);
         .prmess<"modulaire,5>)*)>
%
%enchaine toutes les procedures necessaires pour
%recompiler un programme et remettre a jour
%le catalogue

ss2;.ldef
:&
.def<.mcompile2<@x,@yrface>,
      (@k u*;
       @source=@k;
       .ecrit;
       (.prmess<"modulaire,18>;
        @kenom c &;
        @nyface = @kenom;
        @cat f @nyface;?(@cat r;?,$1;
                         @cat 2;?,$1;
                         .eqval<@cat,@x>;?(@cat u*;$3)
                                          ,@cat u*);
        .prmess<"modulaire,19>;
        .pcat)*;
```

```
       pd;
       .load<@nyface>;
       .prmess<"modulaire,8>;
       (@reponse c&;
        .eqval<@reponse,@oui> /
                  (@yrface:@k;
                   pu;
                   @k 2 1;?(u*;
                            .iface;
                            .prmess<"modulaire,6>;
                            (@reponse c&;
                             .eqval<@reponse,@oui> /
                                      (.reduc;$2);
                             .eqval<@reponse,@non> / $1;
                             .prmess<"modulaire,5>)*;
                           @interface:@k)
                          ,d 3;
                   .leftms;
                   @k c @nyface;
                   @k u*;
                   .eqval<@yrface 2,@k 2>;
                        ?(.prmess<"modulaire,20>;
                          .eqval<@yrface 1,@k 1>;
                               ?(.eqval<@yrface 1 b0,@k 1 b0>;
                                                  ?,.ecrit)
                               ,.ecrit)
                          ,(.ecrit;
                            .querecomp<@nyface>);
          $2);
          .eqval<@reponse,@non> / (@interface : @k;
                                   $2);
          .prmess<"modulaire,5>)*;
       @cat f @nyface;
       @cat u;
       .eqval<@cat b0,@invalidation>;?(@cat b0 c @k b0 3);
       @cat u*;
       @k : @cat;
       .store<@catalog>)>
 %
 %si la nouvelle interface est identique a la precedente
 %(si elle contient exactement les memes declarations),
 %alors il est inutile de recompiler les programmes utilisant
 %cette interface
 %@nyface designe le nom de l'interface courante qui existait
 %@yrface designe l'interface

ss2;.ldef
:&
.def<.verif,
     (.is<@call,@k 1 b0>;?
                        ,(.prmess<"modulaire,4>;
                          $-2);
      .triuse<@k 1 b0 2>)>
 %
 %verifie que l'annotation USE(A,B,C)
 %existe et est correctement structuree
```

```
ss2;.lredef
:&
.def<.iface,
      (.fwd;
       .metex;
       .lab;?(d;u*);
       .val;?(d;u*))>
 %
 %cree l'interface du programme contenu dans @k
 %execute .fwd, .metex; enleve les zones label et value

ss2;.ldef
:&
.def<.fwd<@fwrd,@blocbdnv,@qwert>,
      (@fwrd = @block;
       @fwrd 3 c @forward;
       @blocbdnv = @block;
       @qwert = @lstat;
       @blocbdnv 3 c @qwert;
       2;
       .apl<(.is<@fwrd>;?
                        (@blocbdnv 1 c 1;
                         d @blocbdnv 1 2;
                         d @blocbdnv 1 3;?;
                         d ff @blocbdnv)
             )>;
       u*)>
 %
 %cherche les procedures et fonctions dont l'entete et le corps
 %sont separees; enleve le sous-arbre contenant la procedure et
 %sa definition complete; laisse la declaration dont le corps
 %est: forward

ss2;.ldef
:&
.def<.metex,
      (d 3;
       2;
       .foreach<@block,(.is<@pascal,3>;?
                                        ,(.is<@metasym,3>;?
                                                  ,(d 2;
                                                    3 c @ext1)))>;
       u*)>
 %
 %dans toutes les procedures et fonctions de niveau global,
 %remplace leur corps par: external
 %le probleme des procedures ou fonctions declarees en 2 parties
 %a l'aide de forward est resolu par .fwd

ss2;.ldef
:&
.def<.crintf<@nom,@nomp>,
     (u*;
       .verif;
       .leftms;
```

```
        @nomp = @k;
        @k u*;
        .iface;
        .prmess<"modulaire,6>;
        (@reponse c&;
         .eqval<@reponse,@oui> / (.reduc;$2);
         .eqval<@reponse,@non> / $1;
         .prmess<"modulaire,5>)*;
        (.prmess<"modulaire,21>;
         @kenom c &;
         @nom = @kenom;
         .eqval<@nom,@nomp> / .prmess<"modulaire,22>;
         @cat f @nom;?(.prmess<"modulaire,11>;
                       @cat u*)
                      ,$+1)*;
        .leftms;
        @k c @nom;
        @k u*;
        .eqval<@ookk,@non>;?.sauve,.write;
        .range<@k>;
        @cat f @nom;
        @cat u;
        @cat 2 2 c @nomp;
        @cat 2 b0 c @k b0 3;
        @cat 2 b1 c @k 1 b0;
        @cat u;
        pd;
        @k : @cat;
        .store<@catalog>;
        pu)>
 %
 %@k designe un programme ou une interface pre-existante
 %sur lequel on veut creer une interface

ss2;.lredef
.end
@delete:&
lline]
 -bf
delete *)
@pppp : &
line]
(.polish .pascal) -bf
.mentolin<@catalogue>
$
.xin
```

MERGEX.mentol

```
 %
 %
 %                          PASCAL  MODULAIRE
```

```
%
%                              Valerie MIGOT
%
%
.rec
editeur
:&
.def<.metdef<@lextw,@intfa>,
      (@lextw : @k 1 2;
       @interface f @var;?
             (@k f @var;
              @interface 1;
              (@interface 1;
               @interface 1;
               (@lextw f @interface;?@lextw u
                                  ,(@k f @interface;
                                        ?,(@k : @source;
                                           .prmess<"modulaire,13>;
                                           $-6);
                                     .putdef;
                                     @k u3);
                @interface r)*;?;
               @interface u2;
               @interface r)*;?;
              @interface u*;
              @k u*);
       @interface 2;
       @interface f @block;?(((@lextw f @interface 1 1;?@lextw u
                                ,(@k f @interface 1;
                                      ?,(@k : @source;
                                         .prmess<"modulaire,13>;
                                         $-5);
                                   .putdef;
                                   .eqval<r 2,@forward>;
                                        ?(u;r;
                                          @intfa = @interface 1;
                                          d @intfa 2;
                                          d @intfa 3;?;
                                          ff @intfa;
                                          u;
                                          .putdef);
                                   @k u2);
                               @interface r)*;?);
       @interface u*;
       u*)>
%
%met toutes les variables,fonctions et procedures
%qui apparaissent dans l'interface du programme,
%en def

ss2;.lredef
:&
.def<.merge<@use,@interf,@mdle>,
      (@use : @k 1 b0 2;
       @use 1;?((pd;
                 @interf : @use;
```

```
                .load<@use>;?
                            ,(.prmess<"modulaire,3,@interf>;
                              @cat u*;
                              $-4);
                @mdle : @k;
                pu;
                .merge0<@mdle>;
                @use r)*;?;
               u*);
       .leftms;
       .concate<@nyface,@comp,@np>;
       c @np;
       u*;
       3 1;?u*
           ,.putdef;
       d @k 1 b0)>
 %
 %pour chaque element de la liste ordonnee des interfaces
 %utilisees par le programme dans @k,charge l'interface
 %et insere les declarations de
 %constantes,types,variables,procedures et fonctions qui
 %se trouvent dans l'interface

ss2;.1redef
:&
.def<.merge0<@y>,
      (@k 2 1;?(u*;
                .mergeconst<@y>;
                .mergetype<@y>;
                .mergevar<@y>;
                .mergefproc<@y>)
              ,(@k 2 c @y 2;
                .var;?(.apl<(1;.apl<.putref>)>;
                       r;?((.putref;r)*;?))
                     ,(2 f @block;?((.putref;r)*;?));
                u*))>
 %
 %si @k a une liste de zones non vide,
 %alors on insere les constantes,les types,les
 %variables,les procedures et les fonctions
 %de @m dans le bon ordre
 %et la ou il faut
 %sinon on remplace la liste de zones vide de @k
 %par la liste de zones de @m et on met ref
 %aux variables,procedures et fonctions

ss2;.1def
:&
.def<.mergeconst<@m>,
      (@m f @const;?(.con;?1 jj @m
                          ,(.lab;?i @m
                                 ,(2 1;
                                   j @m));
                    @m u*;
                    u*))>
 %
```

```
%insere dans @k la zone de constantes du module @m,
%si elle existe, avant les constantes de @k s'il y en a
%sinon apres la zone des labels, sinon au debut de la
%liste de zones

ss2;.ldef
:&
.def<.mergetype<@m>,
      (@m f @type;?(.typ;?1 jj @m
                          ,(.con;?i @m
                                 ,(.lab;?i @m
                                        ,(2 1;
                                          j @m)));
                    @m u*;
                    u*))>
%
%insere la zone des types de @m,si elle existe,
%avant les types de @k s'il y en a,sinon apres
%les constantes,sinon apres les labels,
%sinon au debut de la liste de zones

ss2;.ldef
:&
.def<.mergevar<@m>,
      (@m f @var;?(.var;?(1;
                          @m 1;
                          (j @m;
                           1;
                           1;
                           .apl<.putref>;
                           u;
                           r;
                           @m r)*;?)
                         ,(.val;?(j @m;
                                  1;
                                  .apl<(1;.apl<.putref>)>)
                                 ,(2 f @block;
                                     ?(j @m;
                                       1;
                                       .apl<(1;.apl<.putref>)>)
                                      ,(@k 2 s*;
                                        i @m;
                                        r;
                                        .apl<(1;.apl<.putref>)>)))));
                    @m u*;
                    u*))>
%
%insere la zone de variables de @m dans @k,si elle existe et
%met ces variables en ref
%la zone de variables est inseree avant les variables de @k s'il
%y en a,sinon avant la zone de value,sinon avant les procedures
%et fonctions,sinon a la fin de la liste de zones

ss2;.ldef
:&
.def<.mergefproc<@m>,
```

```
    (@m 2 f @block;?(2 f @block;?((j @m;
                                    1;
                                    .putref;
                                    r;
                                    @m r)*;?)
                                  ,(2 s*;
                                    (i @m;
                                    r;
                                    .putref;
                                    @m r)*;?);
                        @m u*;
                        u*))>
 %
 %insere les fonctions et procedures de @m,s'il y en a,
 %avant les procedures et fonctions de @k,sinon a la
 %fin de la liste de zones de @k

ss2;.ldef
:&
.def<.merge0sansref<@y>,
      (@k 2 1;?(u*;
                .mergeconst<@y>;
                .mergetype<@y>;
                .mergevarsansref<@y>;
                .mergefprocsanref<@y>))>
 %
 %merge0 sans ref

ss2;.ldef
:&
.def<.mergevarsansref<@m>,
      (@m f @var;?(.var;?1 jj @m
                        ,(.val;?j @m
                                ,(2 f @block;?j @m
                                            ,(@k 2 s*;
                                              i @m)));
                    @m u*;
                    u*))>
 %
 %fusionne les variables sans les mettre en ref

ss2;.ldef
:&
.def<.mergefprocsanref<@m>,
      (@m 2 f @block;?(2 f @block;?((j @m;
                                    @m r)*;?)
                                  ,(2 s*;
                                    (i @m;
                                    r;
                                    @m r)*;?);
                        @m u*;
                        u*))>
 %
 %fusionne les procedures et fonctions sans les mettre en ref

ss2;.ldef
```

```
:&
.def<.fusionsimple<@pp,@ncm>,
      (.prmess<"modulaire,40>;
       @nom = @call;
       @nom 1;
       @nom c &;
       .load<@nom>;
       @pp = @k;
       @nom c &;
       .load<@nom>;
       @qq = @k;
       @k 2 1;?(u*;
                .mergeconst<@pp>;
                .mergetype<@pp>;
                .mergevarsansref<@pp>;
                .mergefprocsanref<@pp>)
               ,@k 2 c @pp 2;
        .prmess<"modulaire,41>;
        @nom c &;
        .leftms;
        c @nom;
        u*)>
 %
 %fusion de deux programmes

ss2;.lredef
.end
@comp : &
ident]
c
$
.xin
```

UTILITAIRES.mentol

```
 %
 %
 %                          PASCAL  MODULAIRE
 %
 %                            Valerie MIGOT
 %
 %
.rec
editeur
:&
.def<.miseajour<@x,@xx>,
      (@x 1;?,@x:@k;
       .left<@x,@xx>;
       @x u*;
       @cat f @xx;?(@cat u;
                    @cat b0 c @x b0 3;
                    @cat b1 c @x 1 b0;
```

```
                    .is<@call,@cat>;?(@cat u*;$2)
                          ,((@cat u;
                             @cat r;
                             @cat ff @xx;
                             @cat u;
                             @cat b0 c @x b0 3;
                             @cat bl c @x 1 b0)*;?;@cat u*))
                  ,.prmess<"modulaire,10,@xx>)>
 %
 %met a jour dans le catalogue la date et la liste USE du
 %programme ou de l'interface designe par @x

ss2;.lredef
:&
.def<.ecrit<@pmx>,
      (.eqval<@ookk,@non>;?.sauve,.write;
       .leftms;
       @pmx : @k;
       u*;
       .miseajour<@pmx>;
       .cycle<@pmx,@k 1 b0 2>;?,(.tri;
                                 .apl2<@cat,.cycle<1,bl 2>>;
                                    ?,(.prmess<"modulaire,23>;$-3)))>
 %
 %sauve le programme ou l'interface et met a jour le catalogue

ss2;.lredef
:&
.def<.lintf<@prog,.byref<@lintf>>,
     (@lintf = @lident;
      .prmess<"modulaire,46>;
      @kenom c &;
      @prog = @kenom;
      @cat f @prog;?(@cat u2;
                     .is<@ident,@cat 1>;
                        ?,(.prmess<"modulaire,48>;
                           @cat u*;
                           $+3);
                     @lintf s* i @cat 1;
                     @cat r;?(@cat ff @prog;
                                 ?(@cat u2;
                                   @lintf s* i @cat 1;
                                   @cat r))*;?);
      @cat u*;
      d @lintf 1;
      .prmess<"modulaire,47,@prog>;
      @lintf p)>
 %
 %donne la liste des interfaces construites sur un programme donne

ss2;.lredef
:&
.def<.chcat,
      (d @cat;
       .prmess<"modulaire,1>;
       @kenom c &;
```

```
       @catalog = @kenom;
       .pcat)>
 %
 %charge un nouveau catalogue

ss2;.lredef
:&
.def<.aide,
      (.apl2<@aide,p>)>
 %
 %pour appeler au secours

ss2;.lredef
:&
.def<.detail<@n,@command>,
      (.pnnl<@mesapp 3 >;
       @n = @lline;
       @n sl c &;
       .concat<@helpcommand s* ,@n 1 ,@command>;
       .os<@command>)>
 %
 %fournit des informations sur la procedure demandee

ss2;.lredef
:&
.def<.liste,
      (.apl2<@liste,p>)>
 %
 %liste les procedures pour lesquelles il existe des informations

ss2;.lredef
:&
.def<.concate<@ligne,@iidd,.byref<@res>>,
      (.concat<@ligne,@iidd,@res>;
       .coerce<@iidd,@res,@res>)>
 %
 %Le resultat de la concatenation des deux idents est un ident.

ss2;.lredef
:&
.def<.left<@x,.byref<@xx>>,
      ((@x 1/;$)*;
       .is<@ident,@x>;
       @xx : @x;
       @x u*)>
 %
 %si @x designe un programme Pascal, @xx designera son nom

ss2;.lredef
:&
.def<.date<@xln,.byref<@xiddd>>,
      (.coerce<@lchtruc,@xln,@xlch>;
       (d @xlch 16)3;
       d @xlch 13;
       (d @xlch 9)2;
       d @xlch 6;
```

```
       d @xlch 3;
       @xlch 1 j @xlch 5;
       @xlch 1 i @xlch 7;
       (d @xlch 7)2;
       @xlch 1 j @dd;
       .coerce<@iddd,@xlch,@xiddd>)>
 %
 %@xln designe une date de type ligne: 12/13/82  09:44
 %@xid sera D8212130944

ss2;.lredef
:&
.def<.intfbind<@nomc>,
      (.var;?(.prmess<"modulaire,17>;
              $-2)
             ,(.leftms;
               .generenom<@k,@nomc>;
               u*;
               2;
               .os<@fo 4>;
               .foreach<@block,.manip>;
               u*;
               .os<@fo 5>;
               .ecrit))>
 %
 %@k designe une interface dont le nom est deja dans le catalogue

ss2;.lredef
:&
.def<.manip<@bf,@bfa>,
      (.os<@fo 1>;
       1 lp;
       .os<@fo 2>;
       .os<@fo 3>;
       .concat<@car,@bf,@bfa>;
       .concat<@nomc,@bfa,@bf>;
       .coerce<@ext1,@bf,@bfa>;
       3 c @bfa)>
 %
 %

ss2;.ldef
:&
.def<.bind<@x,@bibi>,
      (.concat<@bindage,@x,@bibi>;
       .os<@bibi>)>
 %
 %@x contient le nom d'un programme pascal deja compile
 %sur lequel on veut appliquer la commande exec 'com' pour le
 %'binder'

ss2;.lredef
.end
@liste:&
lline]
```

```
bavard          bind            chcat           checkintf
checkuseintf    crintf          detail          ecrit
fusionintf      fusionsimple    iface           intfbind
lintf           liste           listing         mcompile
metdef          pcat            polish          realize
reduc           saveintf        tri             use
wizard


*)
@helpcommand s* i &
help mentor.mesproc.
@bindage:&
lline]
ec com *)
@chtruc:&
charcst]
´_´;
@lchtruc:&
lchar]
´_´;
@iddd : &
ident]
id
@dd : &
charcst]
´D´;
@fo : &
lline]
fo FICH -tc
ro
interf_1
io attach fich vfile_ FICH
io detach fich
*)
@car : &
charcst]
´$´;
$
.xin
```

MODULAIRE.messages

```
1 $
*1  Donner le nom du catalogue dans lequel vous voulez travailler:
*2  Le catalogue est vide.
```

```
*3  ERREUR: l'interface %1 n'existe pas.
*4  La liste USE est oubliee ou mal structuree
*5  Repondez oui ou non %?
*6  Voulez-vous reduire l'interface globale? (oui/non) %?
*8  Voulez-vous remplacer l'interface existante? (oui/non) %?
*9  Voulez-vous utiliser une interface existante? (oui/non) %?
*10 %1 n'est pas dans le catalogue.
*11 Ce nom existe deja.
*12 %1 est automatiquement conserve.
*13 L'interface n'est plus valide.
*14 Pour obtenir la liste des procedures pour lesquelles on peut %!%
avoir des renseignements faire: %!%
.liste %!%
Pour obtenir des renseignements sur une procedure %!%
appartenant a cette liste faire: %!%
.detail
*15 Cette liste se trouve dans la variable @arecompiler.
*17 L'interface contient une zone de declarations de variables %!%
qu'il faut enlever sinon la commande .bind echoue.
*18 Donner le nom de l'interface qu'il faut utiliser pour %!%
compiler le programme: %?
*19 Ce n'est pas une interface sur ce programme; voici le catalogue:
*20 L'interface n'a pas ete modifiee.
*21 Quel nom voulez-vous donner a l'interface? %?
*22 C'est le nom du programme.
*23 Il y a un cycle dans le graphe des dependances
*24 Les composants suivants ne sont plus valides:
*25 La liste des programmes qu'il faut recompiler est vide.
*26 %1 doit etre declare
*27 Donner la liste des interfaces qu'il faut fusionner:
*28 Donner le nom de l'interface ainsi creee:
*29 Nom de l'interface virtuelle:
*30 Nom du programme modulaire cree:
*31 Le programme courant ne realise pas l'interface citee.
*32 Voulez-vous laisser tous les sous-programmes? (oui/non) %?
*33 Voulez-vous laisser toutes les variables? (oui/non) %?
*34 Voulez-vous laisser tous les types? (oui/non) %?
*35 Voulez-vous laisser toutes les constantes? (oui/non) %?
*36 Voulez-vous laisser %1 ? (oui/non) %?
*38 Liste des objets a enlever de l'interface (types et constantes en dernier):
*39 Liste des objets a laisser dans l'interface:
*40 Noms des deux programmes a fusionner (separes par RETURN):
*41 Nom du programme cree: %?
*42 Voulez-vous enlever tous les sous-programmes? (oui/non) %?
*43 Voulez-vous enlever toutes les variables? (oui/non) %?
*44 Voulez-vous enlever tous les types? (oui/non) %?
*45 Voulez-vous enlever toutes les constantes? (oui/non) %?
*46 Nom du programme: %?
*47 Liste des interfaces sur le programme %1 :
*48 Ce n'est pas le nom d'un programme
*49 La liste USE du programme %1 a ete omise dans le catalogue.
```

3– Les programmes Pascal

```
 (*USE*)
program TREEFRONT_I;
   const
      MAXBUFFERS=16;
   type
      PRIVATE =@INTEGER;
      BUFFERS =array[1..MAXBUFFERS]of CHAR;
      BUFFSIZES=0..MAXBUFFERS;
      TREES   =PRIVATE;
      OPERATORS=0..255;
      VARIABLE=PRIVATE;
      LANGUAGE=@INTEGER;
   var
      ERR:BOOLEAN;
      LANGUE,LOCALANG:LANGUAGE;

   function GETVARIABLE(var NAME:BUFFERS;SIZE:INTEGER):VARIABLE;external;

   procedure SEND(V:VARIABLE;T:TREES);external;

   function RECEIVE(V:VARIABLE):TREES;external;

   function ATOMIDENT(OP:OPERATORS;var NAME:BUFFERS;SIZE:BUFFSIZES):TREES;
    external;

   function MAKEBINARY(OP:OPERATORS;S1,S2:TREES):TREES;external;

   function MAKELIST(OP:OPERATORS;T:TREES):TREES;external;

   function POST(L,T:TREES):TREES;external;

   function HEAD(L:TREES):TREES;external;

   function TAIL(L:TREES):TREES;external;

   function ISEMPTY(L:TREES):BOOLEAN;external;

   function CHILD(N:INTEGER;T:TREES):TREES;external;

   function EQTREE(T1,T2:TREES;L:LANGUAGE):BOOLEAN;external;


 (*USE(TREEFRONT_I)*)
program ORDON(OUTPUT);
   const
      NN      =20;
      P       =100;
   type
      GRAPHE  =array[1..NN,1..NN]of INTEGER;
      LIGNE   =array[1..NN]of INTEGER;
```

```
   CHEMIN  =array[1..P,1..NN]of INTEGER;
var
   N,F,K,NUMCH:INTEGER (*
    N est le nombre de modules contenus dans le catalogue
    *);
   PRED,G,NINIV:GRAPHE;
   CHM:CHEMIN;
   NIV,ORDRE,PT:LIGNE;
   ARBREF,ARBRESULT:TREES;
   BUF,BUFRESULT:BUFFERS;
   VARF,VARESULT:VARIABLE;
   ARBRESINGL:array[1..NN]of TREES;

function NUMERO(var TT:TREES):INTEGER;
    (*telle que arbresingl[numero(tt)]=tt *)
   var
      J:INTEGER;
   begin
   J:=1;
   while not EQTREE(ARBRESINGL[J],TT,LANGUE)do J:=J+1;
   NUMERO:=J
   end (*NUMERO*);

 (*initialise le graphe GG en prenant les donnees dans l'arbre
   @gus ou se trouvent les listes des modules utilises par chaque
   module du catalogue *)
procedure INITIAL(var GG:GRAPHE);
   var
      I,J,K:INTEGER;
      GUS:BUFFERS;
      VARGUS:VARIABLE;
      ARBREGUS,ARBRELIST,POMMIER,ARBREGUS1,ARBRELIST1,ARBRESINGL1:TREES;
   begin
   GUS[1]:='G';
   GUS[2]:='U';
   GUS[3]:='S';
   VARGUS:=GETVARIABLE(GUS,3);
   ARBREGUS:=RECEIVE(VARGUS);
   I:=1;
   while not ISEMPTY(ARBREGUS)do
      begin
      POMMIER:=HEAD(ARBREGUS);
      ARBRESINGL[I]:=CHILD(1,POMMIER);
      if not ISEMPTY(CHILD(2,POMMIER)) then
         begin
         ARBRELIST:=CHILD(2,POMMIER);
         J:=1;
         while not ISEMPTY(ARBRELIST)do
            begin
            ARBRESINGL1:=HEAD(ARBRELIST);
            K:=NUMERO(ARBRESINGL1);
            ARBRELIST1:=TAIL(ARBRELIST);
            ARBRELIST:=ARBRELIST1;
            GG[I,J]:=K;
            J:=J+1
            end
         end
      else J:=1;
      GG[I,J]:=0;
```

```
      I:=I+1;
      ARBREGUS1:=TAIL(ARBREGUS);
      ARBREGUS:=ARBREGUS1
      end;
   N:=I-1
   end (*INITIAL*);

 (*transforme le tableau des predecesseurs en un
   tableau des successeurs
   *)
procedure PREDSUCC(var P,S:GRAPHE);
   var
      I,J,K:INTEGER;
   begin
   for I:=1 to N do
      begin
      PT[I]:=1;
      for J:=1 to N do S[I,J]:=0
      end;
   for I:=1 to N do
      begin
      J:=1;
      while P[I,J]#0 do
         begin
         K:=P[I,J];
         S[K,PT[K]]:=I;
         PT[K]:=PT[K]+1;
         J:=J+1
         end
      end
   end (*PREDSUCC*);

 (*teste s'il y a un cycle dans le graphe donne
   *)
function CYCLE(GR:GRAPHE):BOOLEAN;
   var
      I:INTEGER;
   begin
   CYCLE:=TRUE;
   for I:=1 to N do
      if GR[I,1]=0 then CYCLE:=FALSE
   end (*CYCLE*);

 (*donne tous les chemins entre deux points donnes
   *)
procedure TLC(var DEB,ARR:INTEGER);
   var
      NUMC,D,KK,L:INTEGER;

   function NAPP(var X,H:INTEGER):BOOLEAN;
      var
         M:INTEGER;
      begin
      NAPP:=TRUE;
      for M:=1 to H do
         if CHM[NUMCH,M]=X then NAPP:=FALSE
      end (*NAPP*);
```

```
(*body of TLC*)
  begin
  if G[DEB,PT[DEB]]#0 then
     if DEB#ARR then
        begin
        D:=G[DEB,PT[DEB]];
        CHM[NUMCH,K]:=DEB;
        KK:=K-1;
        if NAPP(D,KK) then
           begin
           K:=K+1;
           PT[DEB]:=PT[DEB]+1;
           TLC(D,ARR)
           end
        else WRITELN('CYCLE ')
        end
     else
        begin
        CHM[NUMCH,K]:=DEB (*for L:=1 to K do WRITE(CHM[NUMCH,L]:4)*);
        if K=N then
           for L:=1 to N do ORDRE[L]:=CHM[NUMCH,L];
        NUMC:=NUMCH;
        NUMCH:=NUMCH+1;
        for L:=1 to K do CHM[NUMCH,L]:=CHM[NUMC,L];
        KK:=K-1;
        if NAPP(D,KK) then
           begin
           D:=G[DEB,PT[DEB]];
           K:=K+1;
           PT[DEB]:=PT[DEB]+1;
           TLC(D,ARR)
           end
        else WRITELN('cycle 2 ')
        end
  else
     if DEB#ARR then
        begin
        PT[DEB]:=1;
        CHM[NUMCH,K]:=0;
        if K#1 then
           begin
           K:=K-1;
           D:=CHM[NUMCH,K];
           TLC(D,ARR)
           end
        end
     else
        begin
        CHM[NUMCH,K]:=DEB (*for L:=1 to K do WRITE(CHM[NUMCH,L]:4)*);
        if K=N then
           for L:=1 to N do ORDRE[L]:=CHM[NUMCH,L];
        NUMC:=NUMCH;
        NUMCH:=NUMCH+1;
        for L:=1 to K do CHM[NUMCH,L]:=CHM[NUMC,L];
        PT[DEB]:=1;
        CHM[NUMCH,K]:=0;
        if K#1 then
           begin
           K:=K-1;
           D:=CHM[NUMCH,K];
```

```
            TLC(D,ARR)
            end
         end
   end (*TLC*);

procedure SUIV(var M:INTEGER);
   var
      I,J,KK:INTEGER;
   begin
   NUMCH:=1;
   for I:=1 to N do
      if G[I,1]=0 then
         begin
         for KK:=1 to N do
            begin
            CHM[NUMCH,KK]:=0;
            PT[KK]:=1
            end;
         K:=1;
         J:=I;
         TLC(M,J)
         end
   end (*SUIV*);

 (*donne les niveaux des points du graphe
   *)
procedure NIVEL;
   var
      I,J:INTEGER;
   begin
   for I:=1 to N do NIV[I]:=0;
   for J:=1 to N do
      for I:=1 to NUMCH do
         if CHM[I,J]#0 then NIV[CHM[I,J]]:=J
   end (*NIVEL*);

 (*si l'ordre entre les points est unique
   *)
procedure ORDUNIQ;
   var
      K:INTEGER;
      ARBRE,ARBRE1,ARBRE2:TREES;
   begin
   ARBRE:=ARBRESINGL[ORDRE[1]];
   ARBRE1:=MAKELIST(73,ARBRE);
   for K:=2 to N do
      begin
      ARBRE:=ARBRESINGL[ORDRE[K]];
      ARBRE2:=ARBRE1;
      ARBRE1:=POST(ARBRE2,ARBRE)
      end;
   ARBRESULT:=ARBRE1;
   VARESULT:=GETVARIABLE(BUFRESULT,7);
   SEND(VARESULT,ARBRESULT)
   end (*ORDUNIQ*);

 (*ordonne les points par niveaux dans l'arbre @ARECOMP
   *)
procedure ORDONNE;
```

```
type
   FORET   =array[2..NN]of TREES;
var
   ARBRESULT1,ARBREIDNIV,ARBRE,POIRIER,POIRIER1,ARBRE1,ARBRE2,ARBRE3,
    ARBRE4,ARBRE5:TREES;
   ARBRES:FORET;
   BUFNIV:BUFFERS;
   I,J,K:INTEGER;
begin
for I:=1 to N do
   for J:=1 to N do NINIV[I,J]:=0;
NINIV[1,1]:=F;
for I:=2 to N-1 do
   begin
   K:=1;
   for J:=1 to N do
      if NIV[J]=I then
         begin
         NINIV[I,K]:=J;
         K:=K+1
         end
   end;
BUFNIV[1]:='N';
BUFNIV[2]:='I';
BUFNIV[3]:='V';
BUFNIV[4]:=CHR(49);
ARBREIDNIV:=ATOMIDENT(1,BUFNIV,4);
ARBRE:=ARBRESINGL[F];
ARBRE1:=MAKELIST(73,ARBRE);
ARBRE:=ARBRE1;
POIRIER:=MAKEBINARY(62,ARBREIDNIV,ARBRE1);
ARBRESULT:=MAKELIST(73,POIRIER);
for I:=2 to N-1 do
   if NINIV[I,1]#0 then
      begin
      if I<=9 then BUFNIV[4]:=CHR(48+I)else BUFNIV[4]:='X';
      ARBREIDNIV:=ATOMIDENT(1,BUFNIV,4);
      ARBRE2:=ARBRESINGL[NINIV[I,1]];
      ARBRE3:=MAKELIST(73,ARBRE2);
      K:=2;
      while NINIV[I,K]#0 do
         begin
         ARBRE4:=ARBRESINGL[NINIV[I,K]];
         ARBRE5:=POST(ARBRE3,ARBRE4);
         ARBRE3:=ARBRE5;
         K:=K+1
         end;
      ARBRES[I]:=ARBRE3;
      POIRIER1:=MAKEBINARY(62,ARBREIDNIV,ARBRES[I]);
      ARBRESULT1:=POST(ARBRESULT,POIRIER1);
      ARBRESULT:=ARBRESULT1
      end;
VARESULT:=GETVARIABLE(BUFRESULT,7);
SEND(VARESULT,ARBRESULT)
end (*ORDONNE*);

(*body of ORDON*)
  begin
  LOCALANG:=LANGUE;
```

```
BUFRESULT[1]:='A';
BUFRESULT[2]:='R';
BUFRESULT[3]:='E';
BUFRESULT[4]:='C';
BUFRESULT[5]:='O';
BUFRESULT[6]:='M';
BUFRESULT[7]:='P';
INITIAL(PRED);
if not ERR then
   begin
   PREDSUCC(PRED,G);
   for K:=1 to N do ORDRE[K]:=0;
   BUF[1]:='X';
   BUF[2]:='X';
   BUF[3]:='X';
   VARF:=GETVARIABLE(BUF,3);
   ARBREF:=RECEIVE(VARF);
   F:=NUMERO(ARBREF);
   if CYCLE(G) then WRITELN('CYCLE pas de point terminal ')
   else SUIV(F);
   if ORDRE[1]#0 then ORDUNIQ
   else
      begin
      NIVEL;
      ORDONNE
      end
   end
else WRITELN('ERREUR')
end. (*ORDON*)
```

Ada 80

Reference Manual for the Ada Programming Language, *Proposed Standard Document* Honeywell, Inc & CII-Honeywell Bull - Juillet 80

Afn 82

Le langage de programmation Pascal, *Spécifications et mise en oeuvre* Normes et techniques - AFNOR - SOL - 1982

Bou 81

J-L.Bouchenez, M.Loyer, F.Prusker, J-C.Sogno, *Legos: un langage modulaire* INRIA - Rapport technique no 1 - Avril 1981

Car 82

L.R.Carter, D.C.Robbins, *An extension to Pascal to provide for separate compilation and encapsulation* Proposition ANSI X3J9/82-073

Cas 81

P.M.Cashin, M.L.Joliat, R.F.Kamel, D.M.Lasker, *Experience with a Modular Typed Language: Protel* 5th Intl.Conf. on Software Engineering - Mars 1981

Don 75

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, J.J.Lévy, *A structure oriented program editor: a first step towards computer assisted programming* INRIA - Rapport de Recherche no 114 - Avril 1975

Don 79

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *Introduction au système Mentor et à ses applications* Actes des journées Francophones sur la Certification du Logiciel - Genève - Janvier 1979 -

Don 80

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *Programming Environnements based on Structured Editors: The Mentor Experience* INRIA - Rapport de Recherche no 26 - Juillet 1980

Don 81

V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, *The Mentor Program Manipulation System* Documentation Multics de l'INRIA - 1981

Est 82

J.Estublier, S.Krakowiak, *Gestion des composants logiciels et composition des systèmes* Projet ADELE - Septembre 1982

Fel 77

S.I.Feldman, *Make – A Program for Maintaining Computer Programs* Bell Laboratories CSTR 57 - 1977

Iso 82

ISO Norme Pascal, *Specification for Computer Programming Language Pascal* ISO/DIS 7185

Jen 78

K.Jensen, N.Wirth, *Pascal User Manual and Report* Springer Verlag - 1978

Lan 83

B.Lang, *Messages and Exceptions in Mentor – Description of a preliminary implementation* Memo Mars 1983

Loy 81

M.Loyer, *Modularité et compilation séparée: Les choix du langage Legos* Thèse de docteur-ingénieur, Université P.Sabatier Toulouse 1981

Mel 80

B.Mélèse, *Manipulation de programmes Pascal au niveau des concepts du langage* Thèse de 3eme cycle, Université Paris 11, juin 1980

Mel 81

B.Mélèse, *Mentor: L'environnement Pascal* INRIA - Rapport Technique no 5 - Octobre 1981

Mel 83

B.Mélèse, *Mentor Rapport – Manipulation de textes structurés sous Mentor* INRIA - Janvier 1983

Mig 83

V.Migot, M.Loyer, *Adjonction de la compilation séparée et de la modularité au langage Pascal à l'aide de Mentor* INRIA - Rapport Technique no 24 - Mai 1983

Mit 79

J.G.Mitchell, W.Maybury, R.Sweet, *Mesa Language Manual – Version 5.0* Xerox Palo Alto Research Center - April 1979

Pas 81

>am>PASCAL>info>pascal.gi.info, *Grenoble University Multics PASCAL Compiler* Documentation Multics de l'INRIA - 1981

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

MIGOT Valérie

SUJET:

Un Pascal Modulaire sous Mentor

RESUMÉ

Cette thèse présente des extensions au langage Pascal pour introduire la compilation séparée et la modularité. Ces extensions ont été réalisées à l'aide de l'environnement de programmation Mentor pour le langage Pascal. Elles sont traitées par un pré-processeur syntaxique qui transforme un programme écrit en Pascal modulaire en un programme acceptable par un compilateur standard. Le pré-processeur permet de définir pour un programme des interfaces (ensemble cohérent d'objets exportés) utilisables par d'autres programmes de l'application. Un catalogue gère les différents composants de l'application (programmes et interfaces) et détecte les conséquences d'une modification d'un composant sur le reste de l'application.

MOTS CLÉS

Pascal
Mentor
Compilation séparée, modularité

ISBN - 2 - 7261 - 0359 - 9